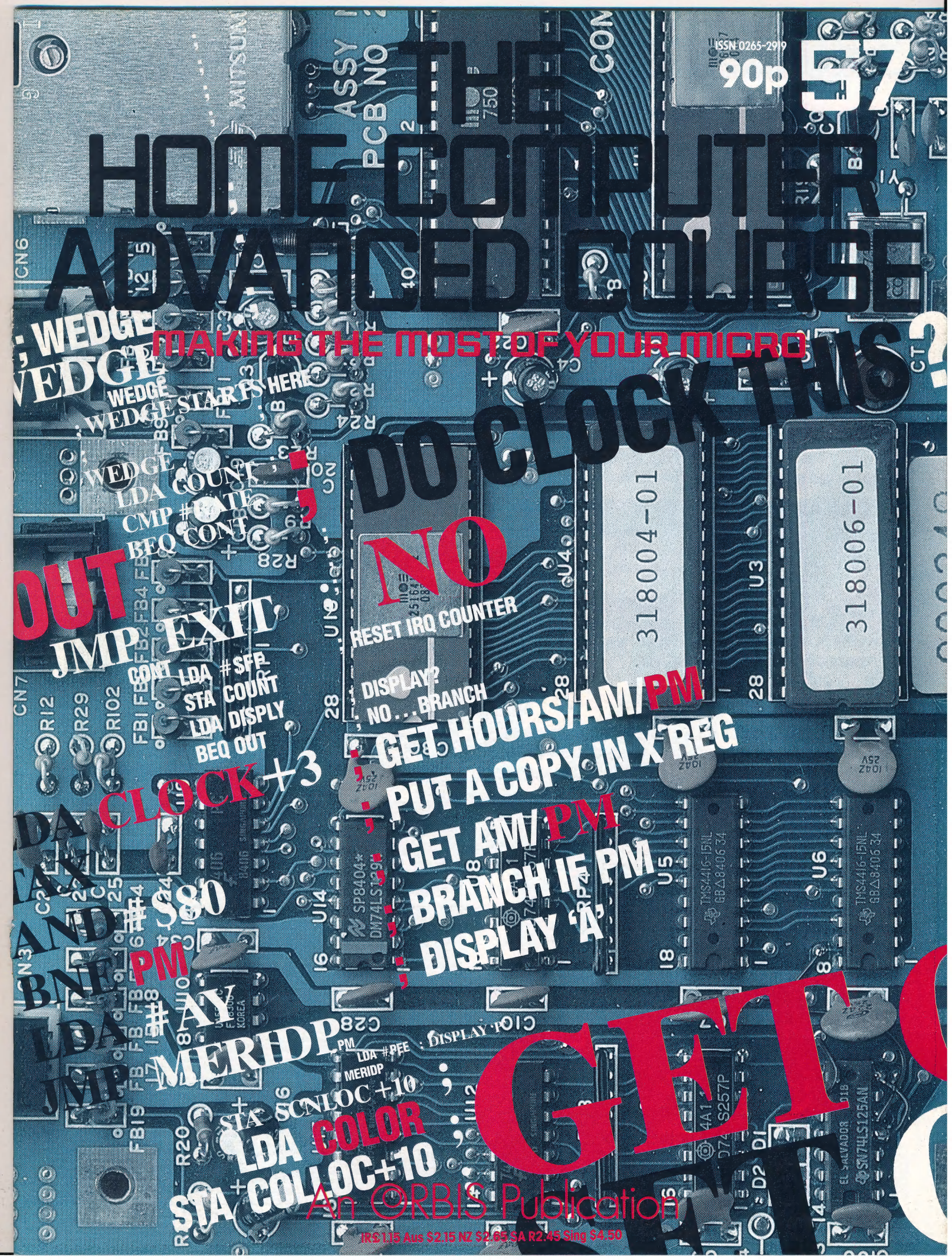
ISSN 0265-2919
90p 57

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

DO CLOCK THIS?



An ©RBIS Publication

IR£1.15 Aus $2.15 NZ $2.65 SA R2.45 Sing $4.50

CONTENTS

APPLICATION

MODERN LANGUAGES We consider some of the major questions surrounding the teaching of programming languages — 1121

HARDWARE

A SLICE OF MEMORY We look at a widely available mass storage system for the Spectrum. The Rotronics Wafadrive is a competitive alternative to Sinclair's own Microdrives — 1129

SOFTWARE

KEYING IN Our series on databases continues with an examination of how primary and secondary keys are used to search and manipulate the database — 1124

COMPUTER SCIENCE

RECORD TIME Pascal's use of 'records' allows us to package data of different types — 1134

JARGON

FROM NAME TO NESTING A weekly glossary of computing terms — 1128

PROGRAMMING PROJECTS

ON THE HIGH SEAS The fifth module in our simulation game gives the program the facility to keep a weekly account of the state of the ship and its crew on our voyage — 1132

MACHINE CODE

AROUND THE BLOCK We begin a series of articles detailing some of the more useful or unusual aspects of the Commodore 64's operating system. To begin, we introduce the concept of a 'wedge', and show how one is used to create a working on-screen clock — 1137

WORKSHOP

UP FOR GRABS This week we show you how to assemble the grab mechanism of the robot arm, and make the rod connections between the servo motors — 1126

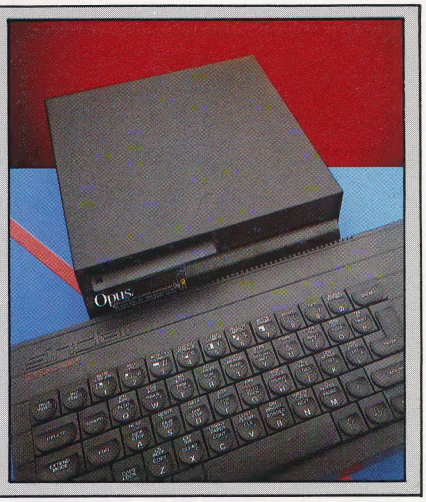
REFERENCE CARD Another section of a comprehensive list of BASIC ROM routine addresses for the BBC Micro, courtesy of Acorn Computers — INSIDE BACK COVER

Next Week

● The Discovery 1 from Opus Supplies is a disk-based system for the Sinclair Spectrum that has the backing of a major High Street retailer. We take a look at what the Discovery has in store.

● Continuing our investigation of the Commodore 64's operating system, we examine input/output management.

● In our series on computers and education, we look at the uses, abuses and misconceptions that occur when introducing new technology into schools.



QUIZ

1) Suggest a reason why 'keys' increase the speed of access in a database.
2) How does the Wafadrive operating system 'know' when to page itself into the Spectrum operating system?
3) What led Borge Christensen to develop the COMAL language?
4) Why is it useful to know the contents of the BASIC pointers?

Answers To Last Week's Quiz

1) The education service provided on Prestel is called 'Schools Link'.
2) Daughter boards are not essential to the running of a micro as these generally contain the circuits for optional utilities.
3) The 'hero' of Knight Lore — Sabre Man — has also been featured in Underwurlde and Sabre Wulf.
4) The Intel 8087 chip provided faster mathematical processing than its companion 8088 main processor

Editor Stephen Cooke; **Art Editor** Claudia Zeff; **Production Editor** Bobby Pickering; **Technical Editor** Steve Colwill; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Steve Malone; **Sub Editor** Jon Kaye; **Contributors** Geoff Bains, Nick Walsh, Dr Antonia Jones, Steve Malone, David Mudd, Anthony Ginn, Steve Colwill; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd: Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Assistant** Alastair Gourlay; **Subscription Manager** Christine Allen; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1985; © Orbis Publishing Ltd 1985; Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Heanor Gate Printing Ltd, Derby**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Price: 90p/IR£1.15. Subscription: 6 months: £26.00. 1 Year: £52.00. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 90p. Subscription: 6 months air: £44.72. Surface: £36.14. 1 year air: £89.44. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 90p. Subscription: 6 months air: £50.18. Surface: £36.14. 1 year air: £100.36. Surface: £72.28. Binder: £5.00. Airmail: £8.25. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/90p. Subscription: 6 months air: £59.54. Surface: £36.14. 1 year air: £119.08. Surface: £72.28. Binder: £5.00. Airmail: £9.50. **SOUTH AFRICA** – Price: SA R2.45. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 90p. Subscription: 6 months air: £64.22. Surface: £36.14. 1 year air £128.44. Surface: £72.28. Binder: £5.00. Airmail: £9.75. **AUSTRALIA** – Price: AusS2.15. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZS2.65. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 5211. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in stirling. Back issues are sold at the cover price, and we do not charge carriage in the UK.
NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.
ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited, Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY CHRIS STEVENS COVER DESIGN BY JULIAN DORR

MODERN LANGUAGES

Q Why is there controversy surrounding the teaching of computer programming to primary school children?

A The traditional role of the child in education is as a passive receptacle for learning. Allow the child an active role and controversy is bound to ensue — so it is hardly surprising that educating children in programming skills is a high-risk area for teachers and students alike.

Computer studies have featured in school curricula since the late 1970s and they have become increasingly popular with both pupils and parents. Surveys of parents in the early 1980s indicated that they regard Computer Studies as the third most important subject in school — after Maths and English — but this enthusiasm is not always shared by members of the teaching profession, nor by those in the computer industry on the lookout for new recruits.

Some teachers, for example, believe that computers would be far better integrated into every area of the curriculum rather than being given separate status as solely the domain of a Computer Studies department. But this has sometimes meant that Computer Science, as a discipline, has suffered — especially with the problems involving resources that most schools are faced with. Despite this, many educationalists still believe passionately in the benefits of programming *per se.*

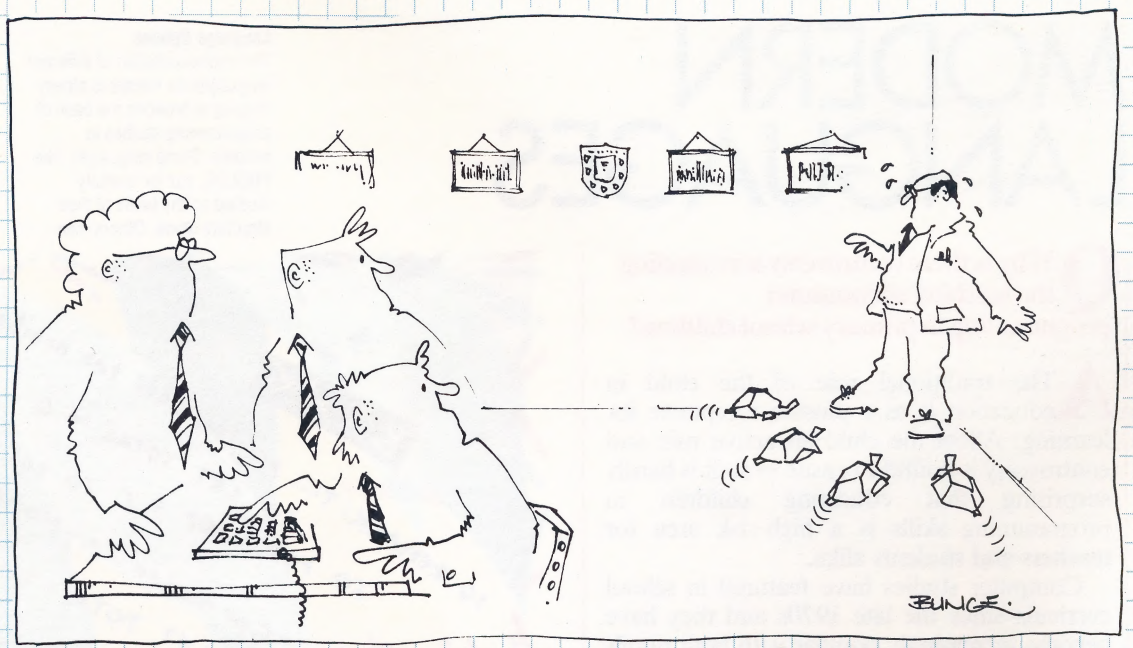
Just because a child will never become a professional artist or writer does not warrant the omission of these studies. As we have seen in this series, young children are helped to learn concepts and skills normally considered beyond their grasp as a by-product of programming, and these 'side benefits' alone must be some justification at least for teaching programming in schools.

The teaching of programming in schools has, until recently, been heavily influenced by the fact that most machines are supplied with BASIC. Because of its abstract content, BASIC is beyond the reach of most children under 12, so in some cases Computer Studies has been restricted to secondary schools (above 6th grade). Furthermore, however useful otherwise, the computing industry generally regards the use of BASIC in education with extreme disfavour because of its unstructured approach and the way in which it can encourage poor programming habits in the beginner. Unfortunately, many teachers involved in

Language Options

The implementation of different languages on micros is slowly helping to broaden the base of programming studies in schools. Some languages, like PROLOG, can be usefully studied on the basis of their structure alone. Others, like COMAL and LOGO, are valuable in offering a more disciplined introduction to programming than BASIC does (which, although easy to learn, is also easy to abuse and can lead to careless programming techniques)

ALAN ADLER



Computer Studies have introduced themselves to the subject by attending short courses on BASIC, thereby propagating its influence despite the availability of other languages like LOGO, COMAL, and PROLOG.

Q If, as widely suggested, the use of BASIC encourages bad programming techniques and sloppy or abstract thought, what are the alternatives?

A The introduction of more structured versions of BASIC — BBC BASIC, for example — has helped to silence some industrial misgivings, but the real interest in educational programming lies in the three languages just mentioned. We have already dealt with LOGO in some detail in the series. It is being actively promoted in education both in Britain and the US by the British Logo Users Group and the Young Persons Logo Association, respectively. All popular micro manufacturers have brought out versions of the language, rescuing it from the obscurity of AI research projects, in which it once seemed destined to remain.

At the State Teacher's College, Tonder, in Denmark, Borge Christensen developed COMAL (COMmon Algorithmic Language). COMAL, according to its creator, is a structured form of BASIC. Christensen and his colleagues found their students' BASIC programs so badly written that they were almost incomprehensible. 'After processing several miles of BASIC programs', he said, 'I realised that something was rotten in the state of Denmark.' Christensen's arguments for COMAL have influenced the Danish government to the extent that they have adopted COMAL as a first programming language for schools. It is also preferred to BASIC in the Republic of Ireland and some interest has developed in Britain. One ambitious project undertaken in a South London secondary school involved students in writing a satellite tracking program in COMAL.

PROLOG (PROgramming in LOGic) has been used by a small group of educationalists working in England. Using PROLOG, a problem expressed in logical terms can easily be translated into a computer program. At Park House School in London, a project was set up to develop materials for use in the teaching of logic as a programming discipline for 10 to 13 year olds. Some became fluent programmers and others coped with the work set in the class, but wrote little of their own. More success was achieved in later classes as computer literacy spread throughout the school. The emphasis of the project has been to teach logic rather than programming.

Q What differences are there in these languages that could determine how a child learns from programming?

A LOGO has been extensively researched in an attempt to answer critics who claim there is little evidence of the benefits children receive by programming in it. (Unfortunately, the relative minority of schools using COMAL and PROLOG prevents studying these languages properly). The Brookline LOGO project was funded by the US National Science Foundation to study children's use of LOGO. The first phase of the project involved having sixth grade (11 year old) students in a computer laboratory and observing styles and activities. The second phase put computers into school, and was partly concerned with developing curriculum materials. An interesting result of the first phase was that all children were successful in their LOGO work, whatever their ability in other subjects. This result has been mirrored in other LOGO projects.

Q What other attempts have been made to study the effects on children who have learned to program?

A The New York Academy of Sciences and three New York City schools set up a project to give training and support for teaching LOGO in school. Computers were installed in classrooms, weekly visits were made by project members, and volunteer teachers attended monthly seminars. The staff estimated the project was successful in almost 95 per cent of the classrooms. The project director, Michael Tempel, remarked on a number of similarities with the Brookline Project. A major consequence of having LOGO in the classroom was a positive interaction among students. Children who had not been successful in other subjects did well with LOGO. 'The positive educational benefit was obvious', he said. 'Kids were engaged in valid intellectual and social processes. We have seen striking changes in kids' relationships to schools and learning.' One result of the project was a recommendation to have a 'magnet school' in the area, to act as a focus for LOGO-centred research.

A 30-month project at the Center for Children and Technology in New York has tried to evaluate the transference of problem-solving strategies to other areas of the curriculum, and to assess the effect of LOGO-oriented work on social interaction. Planning is recognised as an important cognitive skill and studies have shown that children have a poorly developed concept of it, but LOGO's procedural nature encourages breaking programming problems into small units during the planning stage. The Center is focusing on the effects programming has on acquiring these essential learning skills.

One interesting observation has been the way in which computers have encouraged co-operation and interaction amongst the children. 'There was more task-related interaction during computer activity than during other non-teacher-directed classroom activity', noted the program leader. 'Most dramatically, there was more task-related talk around the computer than during other activities'. Similar co-operation and collaboration has been noticed at the Lamplighter School in Texas, to which Texas Instruments donated 50 micros, equipped with TI LOGO.

Q What problems, if any, have hindered the effectiveness of these studies?

A Problems encountered in researching the usefulness of programming in schools are legion. Education authorities are reluctant to set aside much time for programming when the benefits of including it in the curriculum haven't been proved. There is a shortage of funds and hence of hardware. It is also difficult to set up a controlled study, since setting up a control group in the same school means depriving other children of time at the computer, whereas having a control group in another school makes comparison difficult because of different syllabuses and teaching methods. Many teachers are as familiar as children with computers, so there is often no source of knowledge for the children to draw on.

Q Apart from developing more logical approaches to work, what other practical applications can a child expect to encounter after learning to program?

A Once a child has been introduced to the art of programming, the number of potential applications is enormous. One group of children has used turtles to choreograph — having programmed the turtles to move in formation, they then translated the movements of the devices into dance. LOGO's word and list programming features have been used to generate stories in English lessons. Pupils are given a branch illustrating various choices that determine the story's direction. Additionally, LOGO has comprehensive sound facilities. In the same way that SQUARE is used to define PATTERN, NOTE can define CHORD, CHORD can define TUNE, and so on, thereby allowing children to integrate their use of the computer with their study of music.

Q If few schools teach programming, and in different languages, how can there be any widespread benefit?

A In Britain, at least, the steady establishment of LOGO as the programming language for younger children has opened up exciting new areas of curricular activity. But BASIC still reigns supreme at secondary level, and care needs to be exercised to ensure that continued support for LOGO is spread throughout the educational system. Otherwise, the confusion over Computer Studies in schools is liable to be increased as proficient LOGO programmers move up the system and find themselves having to grapple with BASIC. As with computers themselves, the use of them in schools must be standardised to effect any long-term benefits.

Learning to program puts a powerful tool under the control of the pupil, gives children another medium for expression and creativity and encourages them to think in a precise and organised fashion. Finally, it complements many other areas of the curriculum facilitating a much broader understanding of each subject. Although there is clearly a need for much research into the effects and benefits of learning programming, the results achieved so far have been positive and extremely encouraging.

KEYING IN

We have already seen how database managers can organise information into complex structures called records that contain fields of data of various types. But to extract these, we must consider the vital concept of 'keys'. Our series on databases now examines primary and secondary keys.

A *key* is a tool built into the database manager software that helps you to locate a specific record. Consider a car maintenance manual. If you want to adjust the carburettor, the information on how to do it will be tucked away somewhere in the manual. It might be on page 36 (for example), so this page number could be considered as the key to the information you need. Chances are, however, that you will not know that carburettors are dealt with on page 36, so you would turn instead to the index. This will give you the page number you need, enabling you to turn to the right part of the manual without having to leaf through every page.

And so it is with a DBM. Every record in a database file will have a unique record number (known as the *primary key*), but to find a specific record you have the choice of looking at each record in sequence until you find the right one, or, if you know the record number already, going straight to the appropriate record. You also have the choice of using one of the fields as a key (technically known as a *secondary key*). If we had a database on car servicing, we would use the PARTNAME field as the key.

Most DBMs allow specific fields to be designated as 'key fields'. When a field (PARTNAME in this case) has been designated as a key field, the DBM will maintain an internal table of words (character strings) in the specified field, together with the appropriate record number (primary key). When you search for the record dealing with carburettors, the DBM searches through the PARTNAME table until it encounters the character string 'carburettor', sees what the corresponding record number is, and then extracts that record. Here's how you might implement such a scheme in a simple DBM written in BASIC:

```
INPUT 'ENTER KEY FIELD';KEYF$
INPUT 'ENTER WORD TO SEARCH';WRD$
GOSUB 20000 : REM SEARCH SUBROUTINE
PRINT RESULTREC
```

All that is happening here is that one of several arrays has been selected, using KEYF$, and has been searched by the subroutine using the string WRD$ as a search key. If the search subroutine is powerful enough, it will be able to allow for small typing errors and still pick out a likely record. A simple routine like this need not depend on keeping a table of key entries — ordinary searching procedures through all the records will work.

In the past, DBMs that have operated microcomputers have been fairly simple affairs, such as one you could write in BASIC in a reasonably short time. However, a major breakthrough came with the advent of the Sinclair QL, with its 68000 processor. Psion Ltd, using powerful VAX minicomputers, developed Archive for the QL and brought advanced database management capabilities within the reach of the home micro user. To see how Archive works, let's create a trivially simple database on car maintenance with just four records. Each will consist of just two fields, PARTNAME and SERVICE:

```
Carburettor
Remove top and twiddle knob

Oil tank
Lift cap and fill with oil

Battery
Open cap and fill with distilled water

Washer
Open cap and fill with water
```

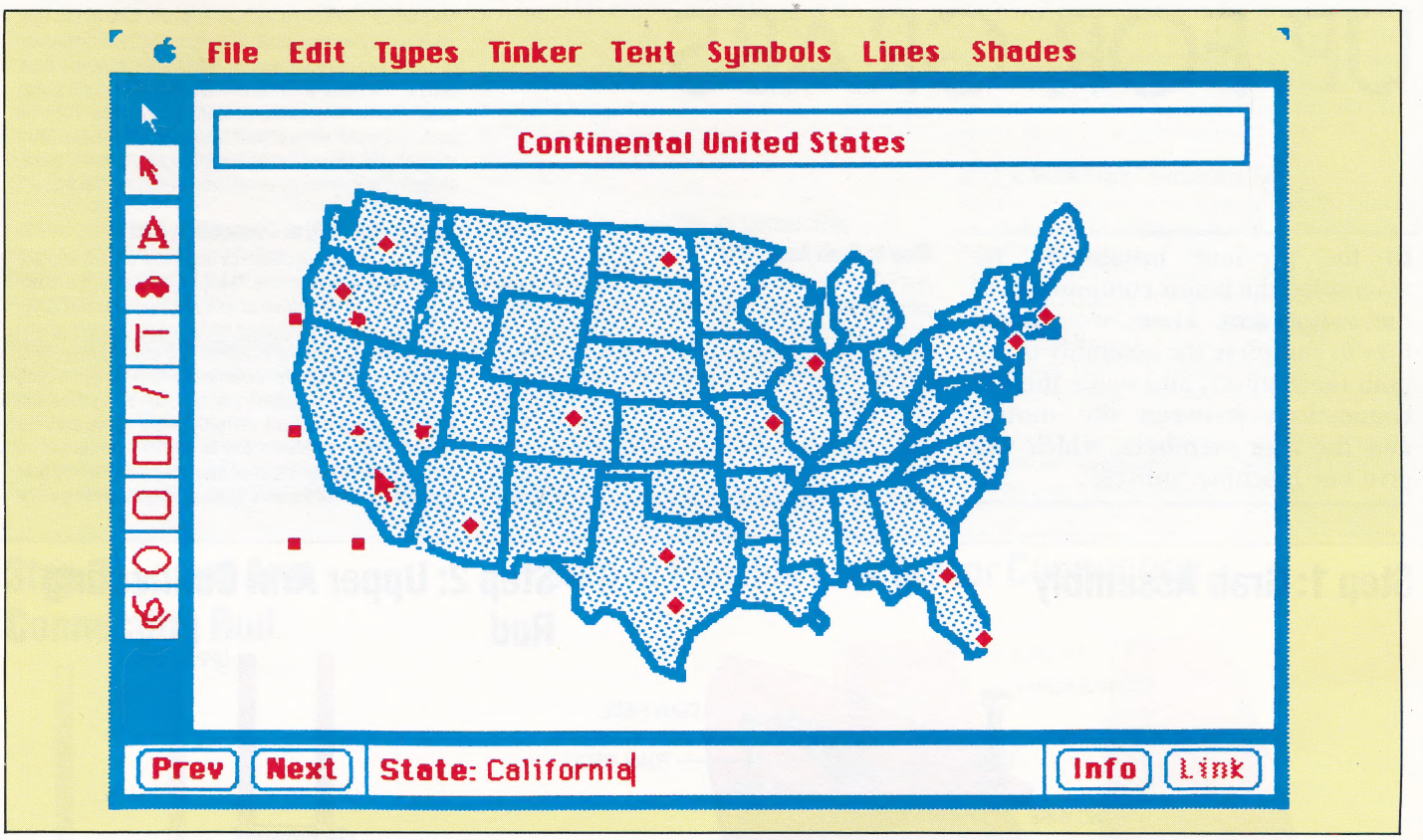
When running Archive on the QL, you are presented with a screen divided into three areas: a command prompt area at the top of the screen, a work area in the centre and a display area at the bottom of the screen. To start a new database file, select the CREATE command. To create our car maintenance database, type CREATE 'CAR' <CR> ('CAR' is the name given to the file). Then enter the names of the fields:

```
PARTNAME$ <CR>
SERVICE$ <CR>
<CR>
```

The $ sign appended to the field names indicates that the field consists of a character string. The final <CR> terminates the process.

To add records to the database, you type the command INSERT<CR>. This displays the field names in the work area and allows you to enter the data for each field. Pressing F5 after a record has been entered correctly adds it to the database file. When all the records you want to enter have been typed in, you can exit the INSERT mode by hitting the Escape key. The file can then be left by typing the CLOSE command.

To search an Archive file for a record, it must first be opened. This can be done by typing either the LOOK command, which allows records to be



examined only, or OPEN, which allows records to be searched and modified. Having OPENed or LOOKed a file, you can enter simple commands such as FIRST (to show the first record in the file), LAST (to show the last record), NEXT (to show the next record) or BACK (to show the previous record).

Searching for a record involves adding an argument to a command, as in FIND 'CARBURETTOR'. This command will search all fields until an entry matching the part in quotes is located. Logical operators can also be used, as in SEARCH PARTNAME$='CARBURETTOR' AND SERVICE$='TWIDDLE'. This will search the file until a record containing the string CARBURETTOR is found in the PARTNAME$ field *and* a string containing TWIDDLE is found in the SERVICE$ field of the same record. The OR operator can be used in the same way, so that a record will be located if the string specified for one field OR the string specified for the other field can be matched.

Records will generally be entered into a database in a haphazard way, but it is frequently necessary to access them or print them out in some kind of order. Suppose, for example, that you have been given the task of entering into a database all the books in a library by author, title, publisher and ISBN (International Standard Book Number). The simplest way to do this would be to work along the shelves, entering the data for each book as you encountered it. By using the sorting capabilities built into your DBM, you could then sort the records alphabetically (by author, title, etc.), or even numerically by ISBN.

It is even possible to combine a primary sort with a secondary sort. Suppose you wanted the file to be printed out in alphabetical order by author, with each author's entries further sorted alphabetically by title. This would enable printouts like this:

```
ADAMS, D. M.
    Duck Rearing for Profit and Pleasure
    Groves Publishing Co.
    0-85527-435-2
ADAMS, D. M.
    Migration Atlas of Ducks and Geese
    Clerihew Books Ltd.
    0-7195-1332-4
ADAMS, D. M.
    Preparing Ducks for the Table
    Groves Publishing Co.
    0-85527-438-6
ADAMS, E.
    Vowel Harmony in Altaic Languages
    University of Chicago Press
    226-52719-0
```

If the field names in Archive were AUTHOR$, TITLE$, PUBLISHER$ and ISBN, sorting the records first by author and then by title would be as simple as:

```
ORDER AUTHOR$;A,TITLE$;A
```

The A in this example stands for 'list by ascending alphabetic order'.

Already, it should be clear that DBMs can make light work of time-consuming tasks such as locating records and re–ordering them in specific ways. We will next look at some of the more advanced facilities built into many DBMs.

The US At Your Fingertips

Filevision was originally developed for the Apple II and III computers, but had to wait until the greater processing power of the Macintosh allowed its remarkable and unique features to be fully appreciated. The program uses the Macintosh's mouse interface to access the database by 'pointing and clicking'. At the 'front' of the program is a sophisticated drawing package that allows you to create a graphics framework for your database. The shapes and symbols you draw are linked into the fields of the database.

We show the main screen for a database of information about the USA. If we point the mouse to indicate California, a simple click gives a brief amount of data about that state; a double click opens a text field that can store a far greater amount of information. The stored data can be linked and sorted in the same way as any normal database. For example, by choosing 'Highlight Some' from the 'Tinker' menu, the program will allow you to specify a search criteria — say, all those states with populations exceeding 10 million — and then display the results by drawing thicker outlines around those states on the map. This information could also be printed out simply by selecting 'Print' after entering the specified criteria

UP FOR GRABS

In the previous instalment, we assembled the major components of our robot arm. Here, we explain how to complete the assembly of the grab mechanism, and make the rod connections between the motors and the arm members, which will give our machine 'muscle'.

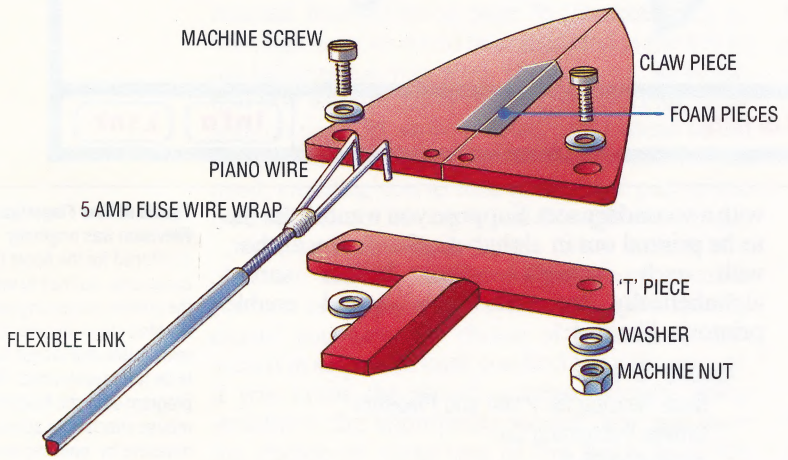
Step 1: Grab Assembly

Drill holes in the centres marked on the 'T' piece and claw pieces to take two machine screws as shown. Glue foam strips into the jaws of the claw pieces as padding. Two further holes should be drilled in the positions marked on the pattern cutouts so that a length of stiff piano wire will fit snugly into the hole. The claw pieces can then be fastened to the 'T' piece using the machine screws and nuts. Insert washers above and below the claw pieces so that when the screws are tightened the claw pieces are free to pivot around the screw. Taking one end of the 500mm flexible link, fasten two 30mm lengths of piano wire to the core cable end by wrapping the piano wire and core cable ends in 5 amp fuse wire and sealing with solder. Bend the other ends of the piano wires downwards, at right-angles, and push into the holes prepared in the claw pieces. The core cable and piano wires should then form a 'Y' shape. With the grab jaws closed, fasten the link casing to the 'T' piece, wrapping with a length of cotton and gluing in place.

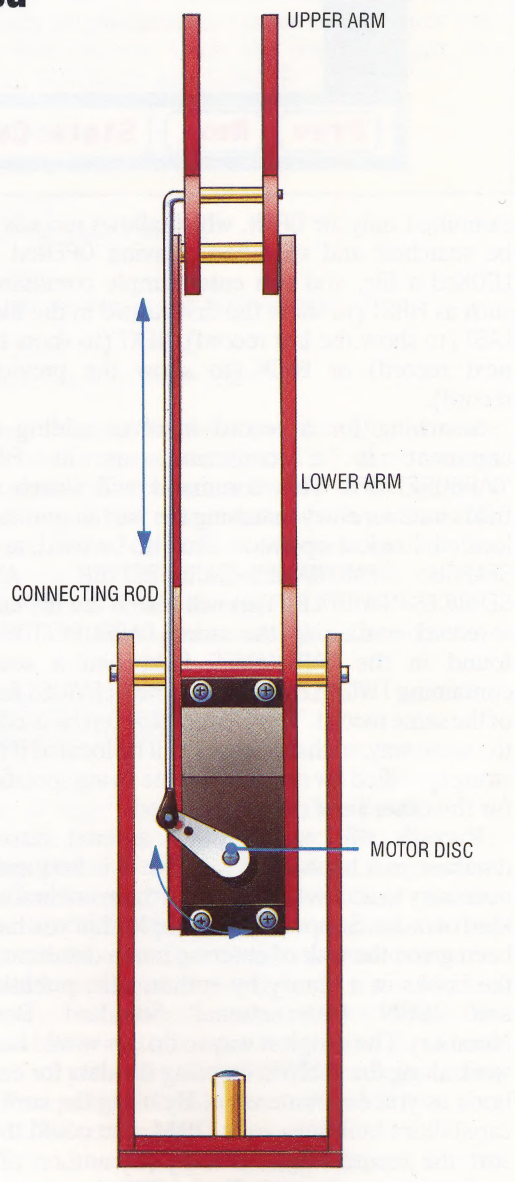
Step 2: Upper Arm Connecting Rod

The upper arm is controlled by the motor already in place on the lower arm assembly. The rotary motion of the motor spindle has to be converted to a push-pull motion at the upper arm. This is achieved by fastening a short plastic arm (or 'horn'), or a 30mm plastic disc to the spindle. Both of these have a hole into which one end of the connecting rod will fit. We will assume that discs are being used, but the system will still work perfectly well if horns are used instead. Place the 30mm disc or horn on the spindle so that when the motor is turned fully anticlockwise, the hole is just to the left of the six o'clock position (thinking of the

Step 1: Grab Assembly



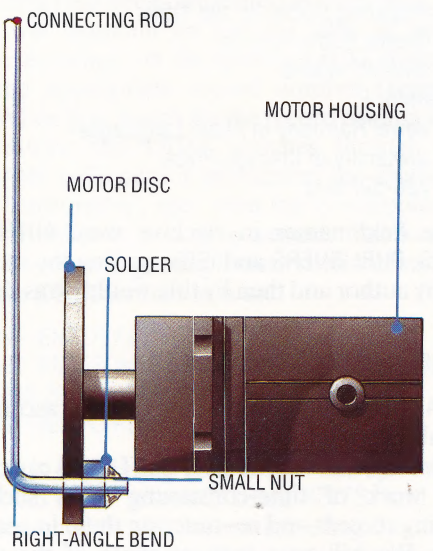
Step 2: Upper Arm Connecting Rod



Connecting Rods To Discs

Special connecting rods may be available from the same source as the servo motors. If they are not, then an ordinary 2mm steel rod is best connected to the motor disc by making a right-angled bend near the end of the rod, pushing this through the hole on the disc and soldering a small nut onto the back of the disc. This is most easily achieved with the disc off the motor spindle.

Where the rods fit into holes in the wooden arm members it is a good idea to line the inside of the hole with a thin piece of brass tubing with an internal diameter just larger than that of the connecting rod. This will reduce abrasion of the wood as the arm moves



KEVIN JONES

motor disc as a clockface with 12 o'clock vertically upwards). The square drive on the spindle should make this position easy to achieve.

Now take a length of 2mm diameter steel rod and make right-angled bends in each end so that the rod fits through the hole in the motor disc or horn and the two holes prepared for it in the upper arm pieces, near the elbow joint. The length of the rod should be such that the lower and upper arms form a 90° angle at the elbow when the motor is turned fully anticlockwise. Check the action of the arm by slowly rotating the motor disc clockwise, by hand. When you are happy with the push-pull action of the rod, glue or solder the rod in place on the motor disc.

Step 3: Lower Arm Connecting Rod

Mount both remaining motors onto the main body, as shown. The right-hand motor controls the movement of the lower arm. Turn the motor so that it is fully anticlockwise (as viewed from the disc side of the motor) and position the motor disc in such a way that the hole in the circumference of the disc is just to the right of the six o'clock position. Fasten a length of 2mm steel rod between the motor disc and the holes in the lower arm near the shoulder joint, as before. The length of the rod should be such that the lower arm is approximately horizontal with the motor turned fully anticlockwise. Again, check the action of the lower arm by rotating the motor disc by hand and glue or solder the connecting rod to the motor disc when satisfied.

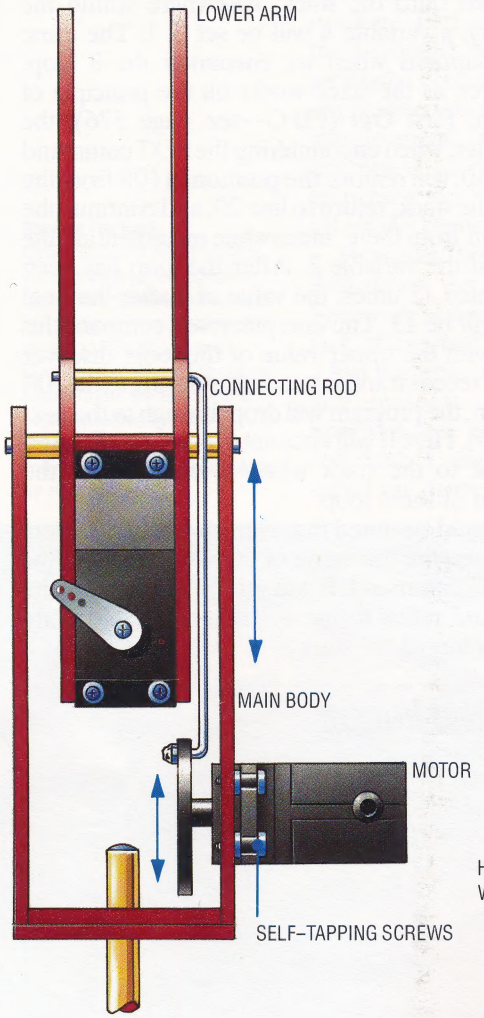
Step 4: Flexible Link Motor Connection

The left-hand motor mounted on the main body is used to open and close the grab by pushing or pulling the core cable in the flexible link. Start by drilling a hole through the main body cross piece so that the flexible link can be threaded through from behind, as shown. Glue the outer casing of the link into this hole. When dry, pull the core cable through so that the grab jaws are tightly closed and fasten the end of the cable to the motor disc. The hole in the circumference of the disc should be at 'nine o'clock' when the motor is turned fully anticlockwise. The cable end can be attached to the motor disc in several ways. It is best to mount a small cable grip on the disc. Alternatively, a 90° bend can be made in the end of the cable and the end pushed through the hole in the disc and glued or soldered in position. No matter how you make this connection, it is important that the motor can push and pull the cable, as it rotates, so that the grab jaws open and close.

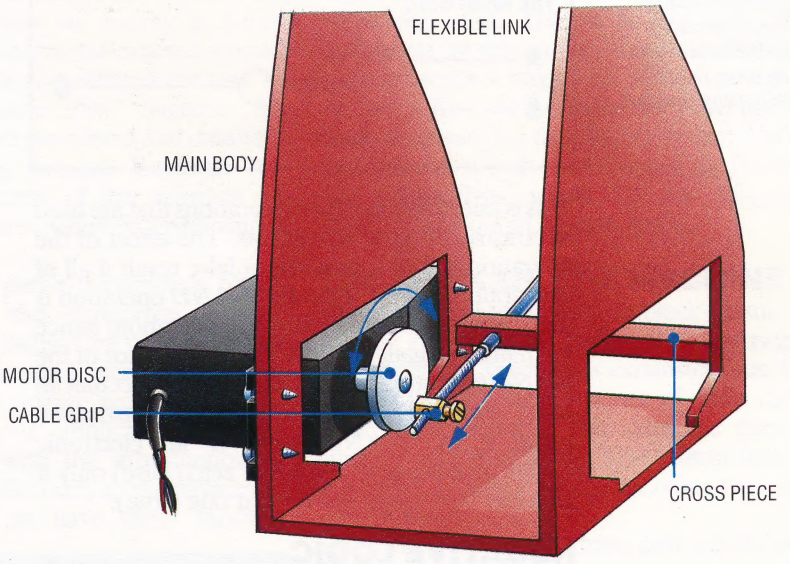
Step 5: Main Body Connecting Rod

The final connection to be made is from the motor mounted in the base box (see page 1096) to the main body assembly. First drill a small hole in the base of the main body (you will need to slide the assembly off the steel dowel to do this). The hole should be positioned — as shown — in line with the pivot, near the right-hand edge (as viewed from above) of the main body base. After the hole has been drilled, place the main body assembly back on the steel dowel and rotate to the two o'clock position, as shown. Turn the motor fully anticlockwise and position the disc so that the hole on the circumference is just to the right of six o'clock. Fit a connecting rod as before and check that the main body is free to rotate anticlockwise.

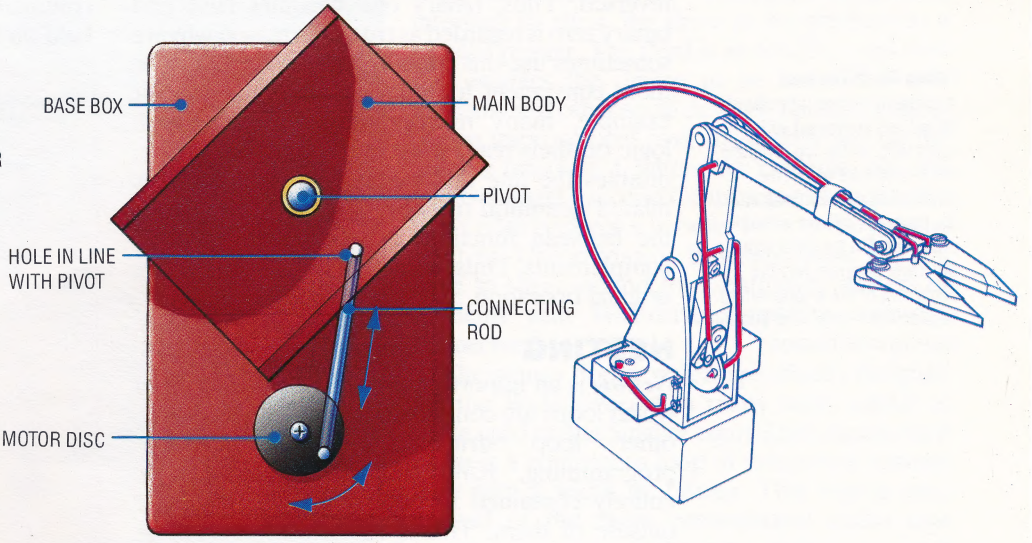
Step 3: Lower Arm Connecting Rod



Step 4: Flexible Link Motor Connection



Step 5: Main Body Connecting Rod

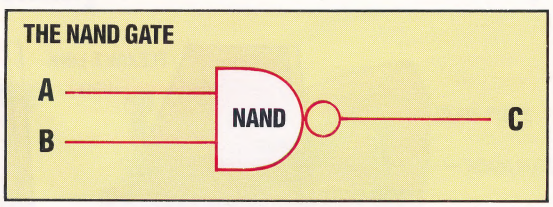


NAME

In computing, as in most areas, a *name* will be allocated to something to make identification easier. Names can be assigned, within a language or operating system, to files, procedures, variables, systems, functions and spreadsheet regions, amongst other things. The reason for naming such objects, apart from making them easier for us to understand, is that once a file or variable has been named, there is no need for you to keep track of its location in memory. The computer will keep tabs on a named object by assigning an address to it, but this can mean little or nothing to the operator. Often a computer will move files around in memory or disk in order to make the best possible use of the available space. However, simply by calling the name, we can recall the file without having to trace its current position.

NAND

Negative AND
The symbolic representation of a NAND gate consists of an AND symbol with a small circle on the output line to indicate the negative condition



LIZ DIXON

This is one of the Boolean operations that are used to construct logical statements. The effect of the operation is that it produces a false result if all of the inputs are true. Thus the *NAND* operation is the logical opposite of the AND operation; hence the name, *N*egative *AND*. The importance of the NAND operation can be judged by the fact that all other Boolean operations can be constructed using it. A 'NAND gate' is an electronic device producing an output of zero (false) only if all of its input lines are held at one (true).

NEGATIVE LOGIC

As its name suggests, *negative logic* is a system whereby the normal conventions of logic are reversed. Thus, binary one becomes false and binary zero is regarded as true. The reason why we sometimes use this form of logic is that it can be more convenient for some types of circuits. For example, many microprocessors use negative logic on their read/write lines where binary one disables the line and binary zero enables it. A related definition of negative logic occurs where the Boolean functions act as though they are complements. Thus the output is the reverse of its original operation.

Falling Out Of The Nest
A correctly nested loop should be entirely contained within an outer loop. In the top program, the Y-loop falls within the boundaries of the X-loop and the program will run without problems. On the other hand, the lower program has the beginning of the X-loop falling outside the Y-loop, and this will cause an error message

NESTING

Nesting is an important programming concept in which loops are contained in their entirety within other loop structures. Thus, in BASIC programming, FOR . . NEXT loops should be entirely contained in other loops and not end outside of them. This is of vital importance in producing orderly code, otherwise the program will at best generate an error message; at worst it could produce a disastrous program crash. When the computer encounters the FOR keyword, it stores the position of the FOR command on the stack. When the program arrives at the associated NEXT command, the computer takes the position of FOR from the stack and returns the program to the point at which it encountered the command. If the loop is not nested, the computer is liable to take the wrong position off the stack and restart the program interpretation from an unintended point with unforeseen results.

Nesting is most commonly used when we wish to perform several operations a number of times. Take, for example, a short program to print out the twelve multiplication tables:

```
10 FOR A=1 TO 12
20 FOR B=1 TO 12
30 PRINT A;"x";B;"=";A*B
40 NEXT B
50 NEXT A
```

First, we begin the A loop. The position of FOR will be saved onto the stack. Elsewhere within the memory, a variable A will be set at 1. The same thing happens when we encounter the B loop. However, as the stack works on the principle of First In, First Out (FIFO—see page 576), the computer, when encountering the NEXT command at line 40, will remove the position of FOR from the top of the stack, return to line 20, and continue the program from there, meanwhile incrementing the value of the variable B. After the loop has been performed 12 times, the value of B after the final NEXT will be 13. The computer will compare this value with the upper value of the loop, discover that it exceeds it and instead of returning to the FOR position, the program will drop through to the next line (50). Here it will encounter another NEXT and go back to the stack where it will retrieve the position of the A loop.

It should be noted that many BASIC interpreters do not require the name of the variable to follow the NEXT command. It will simply assume that the command refers to the position that is currently held on top of the stack.

Nested Loops

```
5  REM Correctly nested loop
10 FOR X=0 TO 6
20 FOR Y=0 TO X
30 PRINT X;":";Y;"";
40 NEXT Y
50 PRINT
60 NEXT X
```

Y-LOOP, X-LOOP

```
5  REM Incorrectly nested loop
10 FOR X=0 TO 6
20 FOR Y=0 TO X
30 PRINT X;":";Y;"";
40 NEXT X
50 PRINT
60 NEXT Y
```

X-LOOP, Y-LOOP

A SLICE OF MEMORY

The initial lack of a reliable, fast storage medium has meant that the Sinclair Spectrum has generally been overlooked as a machine for serious applications. The introduction of Sinclair's own Microdrive system, however, did not deter independent companies from producing storage systems such as the Rotronics Wafadrive.

Although the Sinclair Spectrum has become one of the most popular home microcomputers, it has been criticised in many quarters as being unsuitable for the 'serious' user. This has led to the Spectrum being dismissed as merely a games machine. Part of the problem has centred around the keyboard, which has not allowed the user to seriously consider using the machine for applications such as word processing and database management. Recently, Sinclair Research has attempted to defuse some of this criticism by launching the Spectrum+, which is fitted with a QL–style keyboard.

However, this is only part of the problem. Other difficulties hounding the supporters of the Spectrum as a serious machine are the computer's lack of standard interfaces and, more importantly, a fast and reliable mass storage system — vital to any business or serious application for the hobbyist. Of course, with the introduction of Interface 1 and the Microdrive, the Spectrum could at least claim to have these facilities available. But the suspicion remained that the Microdrive was slow and unreliable. Furthermore, although the software base on cassette for the Spectrum is enormous, very little software has made the transition to the medium used by the Microdrive. In a situation like this, third party suppliers tend to step into the gap and produce alternatives. Here we look at the first of two contenders in the race to dominate the Spectrum mass storage market — the Wafadrive from Rotronics. In the next instalment we will look at the Discovery 1 from Opus Supplies.

Unlike Interface 1 and the accompanying Microdrive units, the Rotronics Wafadrive is an 'all-in-one' unit. That is to say that both the peripheral interfaces and the mass storage units are enclosed in a single box. The advantage of this system is that the units do not have the trailing leads necessary for the Sinclair system, but it does mean that they lack something of the flexibility of the Microdrives, which can be daisy-chained together to expand storage space.

CHRIS STEVENS

Room In The Back
Unlike Sinclair's own system of mass storage for the Spectrum, the Rotronics Wafadrive comes in a single box containing twin drives, an RS232 port and a Centronics interface. The ribbon cable fits into the Spectrum's edge connector, and this positions the drives conveniently above the keyboard for easy access

THE LOOK OF THE MACHINE

The Wafadrive is encased in black plastic with a 35-way ribbon cable ending in a cartridge slot fitted onto the Spectrum's expansion bus. On the front of the unit there is a pair of wafer cartridge drive slots. Between the slots are three light emitting diodes (LEDs). The central light is the power-on indicator, while the other two indicate drive activity.

On the rear of the drive unit are three edge connectors. On the left is a parallel expansion bus to allow Interface 2 to be connected. The centre edge connector is a Centronics-compatible interface to allow the drive to be attached to a parallel printer. The third is an RS232 serial port that enables the device to be interfaced with modems and other serial devices. These interfaces are an improvement over those provided on the Sinclair Interface 1, where, for example, one still has to connect a second Centronics interface to the unit's edge connector in order for it to run parallel printers. Unfortunately, however, users will still have to shop around for Centronics printers or modems with Wafadrive-compatible cartridge connectors.

The stringy floppy wafers specifically designed for the Wafadrive are in many ways similar to those used on the Sinclair equivalents. Inside each wafer is a continuous loop of video-type cassette tape with a width of 1.8mm. This tape is used instead of the more conventional audio tape because of its improved endurance and information storage capabilities. Once formatted,

this tape can contain approximately 128 Kbytes of data, although Rotronics has also made 64 and 16 Kbyte cartridges available.

The cartridges themselves are approximately twice the width of the Sinclair wafers, although they are of similar length and breadth in their protective boxes. This gives the Rotronics cartridges an appearance of miniature cassettes. The Wafadrive cartridges do not need protective casings, as the delicate tape is protected by an automatic sliding cover, similar in design to that on the Sony 3½in microfloppy disks, although the Rotronics protection is made out of plastic instead of metal. On the left side of the wafer is a write protect tab, which can be snapped off. Of course, this tab cannot be replaced once it is broken off and users will have to find some other method of re-enabling their cartridges.

The commands used by the Wafadrive are more or less identical to those used on the Sinclair Microdrives. In both systems the command is followed by a *, indicating that the external storage device is to be accessed. Examples of this usage are SAVE *, LOAD * and VERIFY *. However, the Wafadrive system does have slight differences because there are always two Wafadrives present, as opposed to the numerous Microdrives that could be in the system. For example, when formatting a Sinclair wafer one uses the command FORMAT 'm' ;0;"name", where 'm' ;0 refers to the number of the Microdrive being used. When using the Wafadrive, the command is altered to FORMAT * 'a:name', with the a: referring to the name of the drive in use. Note that with the Wafadrive, there can only be a section a: or b:, whereas on the Microdrive the number can be from zero to seven.

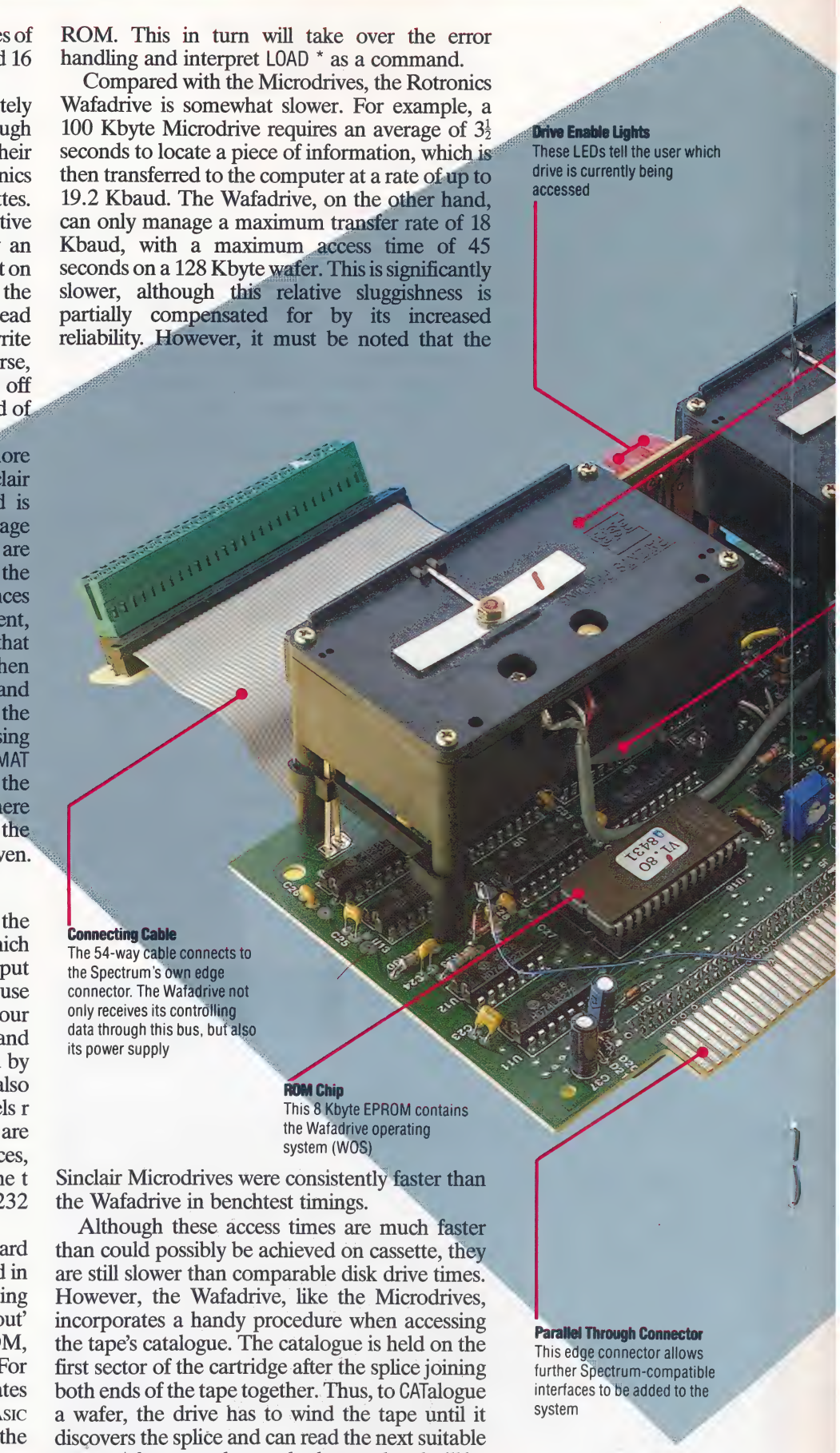
THE 'STREAM' SYSTEM

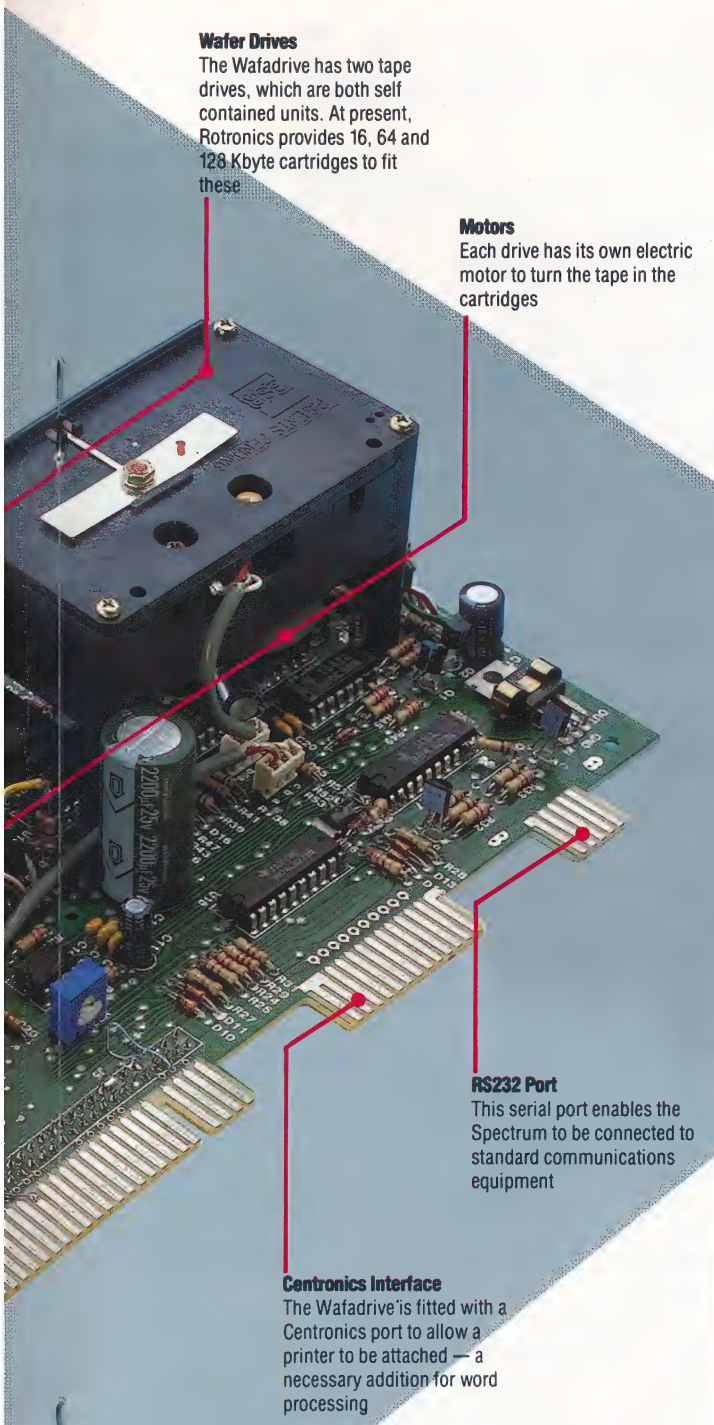
The Wafadrive also takes advantage of the 'stream' system used on the Spectrum, in which there are 16 streams set aside for input/output management. Some of these are reserved for use by the screen and printer. However, channels four to 15 are available to other peripheral devices, and output streams to the Wafadrive are accessed by use of the OPEN # command. The Wafadrive also adds two extra streams to the system. Channels r and c (these letters can also be capitalised) are reserved for the RS232 and Centronics interfaces, respectively, and their usages are similar to the t and b channels — used when accessing the RS232 port on Interface 1.

There is an eight Kbyte ROM on board containing the extended BASIC commands used in controlling the system. This Wafadrive Operating System (WOS) is able to function by 'paging out' the lower eight Kbytes from the Spectrum's ROM, in much the same way that Interface 1 does. For example, the command LOAD * actually generates an error on the Spectrum; thus, when the BASIC interpreter encounters this command on the screen, it will call the error-handling routine. However, this call command will be intercepted by the WOS, which will then page in the Wafadrive ROM. This in turn will take over the error handling and interpret LOAD * as a command.

Compared with the Microdrives, the Rotronics Wafadrive is somewhat slower. For example, a 100 Kbyte Microdrive requires an average of 3½ seconds to locate a piece of information, which is then transferred to the computer at a rate of up to 19.2 Kbaud. The Wafadrive, on the other hand, can only manage a maximum transfer rate of 18 Kbaud, with a maximum access time of 45 seconds on a 128 Kbyte wafer. This is significantly slower, although this relative sluggishness is partially compensated for by its increased reliability. However, it must be noted that the Sinclair Microdrives were consistently faster than the Wafadrive in benchtest timings.

Although these access times are much faster than could possibly be achieved on cassette, they are still slower than comparable disk drive times. However, the Wafadrive, like the Microdrives, incorporates a handy procedure when accessing the tape's catalogue. The catalogue is held on the first sector of the cartridge after the splice joining both ends of the tape together. Thus, to CATalogue a wafer, the drive has to wind the tape until it discovers the splice and can read the next suitable sector. After several seconds, the tape head will be past the catalogue sector. However, should the CAT command be entered again, instead of winding the


Drive Enable Lights
These LEDs tell the user which drive is currently being accessed
Connecting Cable
The 54-way cable connects to the Spectrum's own edge connector. The Wafadrive not only receives its controlling data through this bus, but also its power supply
ROM Chip
This 8 Kbyte EPROM contains the Wafadrive operating system (WOS)
Parallel Through Connector
This edge connector allows further Spectrum-compatible interfaces to be added to the system


Wafer Drives
The Wafadrive has two tape drives, which are both self contained units. At present, Rotronics provides 16, 64 and 128 Kbyte cartridges to fit these
Motors
Each drive has its own electric motor to turn the tape in the cartridges
RS232 Port
This serial port enables the Spectrum to be connected to standard communications equipment
Centronics Interface
The Wafadrive is fitted with a Centronics port to allow a printer to be attached — a necessary addition for word processing

whole of the tape round again, the drive will move for only a fraction of a second before displaying the catalogue again — it will be held in RAM once it has been called. Thereafter, the WOS merely checks to see whether the same wafer is inserted by looking at the next sector. If it is, the WOS will display the catalogue it already holds in RAM.

In keeping with the idea of implementing a mass storage system allowing the Spectrum to be used for more serious applications, Rotronics has included the Spectral Writer word processing program in the package. This is a relatively comprehensive system making full use of the Wafadrive. Functions such as reform paragraph, insert words and delete lines are called by using the Spectrum's Symbol Shift key in conjunction with other keys. Other functions that access files held on wafer can be obtained by means of the options command: these include SAVE and LOAD text files from cassette or from Wafadrive. Spectral Writer is a fine word processor, although it does not enable you to set the line length on the screen. It is perhaps unfortunate that even when using a Spectrum+, the quality of the keyboard somewhat diminishes the Spectral Writer's efficiency.

Of course, what makes or breaks any storage medium on a computer is the willingness of software houses to support it. At the moment this seems to be a serious drawback to the Wafadrive's success, since none of the major software houses are producing their programs on Wafadrive cartridges (this is also a problem that has been encountered by Sinclair itself). However, all is not lost for the Wafadrive user. At least one company is now producing a program that enables you to dump commercial software onto Wafadrive cartridges. This means that users will be forced to buy both the commercial cassette and a wafer to transfer it to, but it may be a small price to pay for the vastly improved access times that are produced.

Another minor difficulty of the Wafadrive is the edge connectors on the back of the machine. Because they are non-standard, users must be willing to convert the interfaces themselves, or else they will have to look hard for peripherals with suitable connections. But although the Wafadrive does have its drawbacks, it is a finely made machine, and certainly a viable alternative to the Interface 1 and Microdrive provided by Sinclair Research.

ROTRONICS WAFADRIVE
PRICE
£129.95 inc VAT
DIMENSIONS
230×110×80mm
INTERFACES
RS232 serial port, Centronics parallel interfaces, Spectrum edge connector
FORMAT
Continuous loop stringy floppy wafers
CAPACITY
16, 64 and 128 Kbyte wafers available
SPEED
Transfer rate: 16 Kbaud; maximum access time: 6.5 sec (16K), 45 sec (128K)

Wafadrive Cartridge
The Wafadrive cartridge is roughly twice the size of the Sinclair Mircodrive's equivalent. The wafer resembles a conventional cassette in its proportions, but the tape inside, is fixed in a continual loop. This means that the tape does not have to be rewound to access data that has gone past the read/write head

CHRIS STEVENS

ON THE HIGH SEAS

Now that we have set sail on a course for the New World, we must keep a weekly account of the crew and ship. The Captain's Log module will cater to these needs, as well as inform us of the unpleasant realities of losing crew members due to malnutrition.

During the voyage to the New World, the ship's speed is affected by the number and strength of the crew. A large, healthy crew does not increase the ship's speed, but when the combined strength rating is lowered below a certain point, inefficiency increases the journey time. If the entire crew dies, then, of course, the game is over.

One of the functions of the Captain's Log is to inform the player of any crew member's death, if a provision runs out and to display a revised estimate of the length of the remaining journey. A new variable, EW, is created on line 41. It represents the extra journey time in weeks — or fractions of weeks — which will be added to the original eight weeks if the cumulative crew strength falls below a certain level. So that the journey length can be displayed and incremented in integers without losing the fractional part of EW, extra journey time is calculated and stored in EW as a real number, but added to JL as an integer using INT.

The programming for the end of week report is in the subroutine at 5300. There will be a large section of program dealing with contingencies during the journey, which will be added in future modules, and we'll begin the programming for this in the next instalment. These events will also affect the amount of provisions, strength and number of the crew and length of journey and will therefore influence the values printed in this instalment's section.

The first task of the end of week report subroutine is to set up a loop at line 5325 to check if any of the crew has died. It looks at the second element of the crew array, TS(,), for each of the 16 possible crew members, and determines whether or not the strength rating is set to -999, which represents death.

The deceased are added to X, which is the weekly count of the victims. If the value in the array is not equal to -999, the next count of the loop is implemented. After recording a loss, line 5340 resets the strength and type elements to 0, to avoid recording the death in the following end of week report. The sailor is then subtracted from the crew total CN. If no one has died, indicated by X remaining at 0 at the end of the loop, the program goes to 5400, where provisions are checked. If all the crew are dead, the program prints 'All the remaining crew died. Your ship drifts helplessly on the empty ocean. Game over.' If some members of the crew are still alive, then the number of deceased is printed at line 5390.

Consumed provisions are recorded in the same way as crew members who have succumbed. The program checks if any provisions have been exhausted and sets a loop for 1 to 4 — representing each provision type. If a provision has been exhausted, its value in array PA() will be -999. Set during the 'issuing provisions' routine, line 5415 checks PA() and the program informs you that 'You have run out of . . .' that particular item if the value is in fact -999.

The element in PA() is then reset to 0, to avoid reprinting the message at the end of the following week. If more than one provision has been exhausted, 'AND' is printed before its name. It is possible that an exhausted provision may be replenished in the contingencies section of the game. If so, the relevant element in PA() corresponding to that provision will be reset, and will feature in all records as normal.

Additional journey time is calculated by going through the crew strength array, accumulating the total strength and comparing it to the minimum optimum strength. To continue the voyage at maximum efficiency, the total crew strength must be at least 800. This could be a crew of eight at full strength, or 16 at half strength, and so on. Line 5465 sets up a loop for each element in the crew array and 5470 accumulates the strengths in X.

Line 5480 checks if the total strength is greater than 799. If so, the journey time will not be increased, and the program goes to 5494, which asks for a key to be pressed to start the next week.

If the strength is less than 800, the formula on line 5489 calculates the extra journey time. It subtracts the cumulative crew strength from 800, divides it by 800 and adds that to the extra journey time. The increase is stored in EW as a real number. Line 5490 adds the integer part of EW onto the journey length, so the remaining journey time is not displayed in fractions of a week. The accumulated value of EW is added to the journey time each week. To ensure that a dead crew member's strength rating is not decreased by WF in the subroutine at line 9300 given in the previous instalment, insert the following line:

```
9315 IF TS(S1,2)=0 THEN 9340
```

As the ship sails to the New World, various events will influence the strength of the crew. The only factor thus far in the program to affect their strength is diet. Because food is shared equally, an inadequate diet lowers the strength of all members

Module Five: Captain's Log

End Of Week Report

```
5300 REM END OF WEEK REPORT
5305 PRINTCHR$(147)
5310 S$="          CAPTAINS LOG*":GOSUB9100
5312 S$="          -----------*":GOSUB9100
5314 GOSUB9200
5316 PRINT:PRINT"          END OF WEEK";WK
5318 GOSUB9200
5320 X=0
5325 FORT=1TO16
5330 IFTS(T,2)<>-999THEN5350
5335 X=X+1
5340 TS(T,1)=0:TS(T,2)=0
5345 CN=CN-1
5350 NEXT
5355 IFX=0THEN5400
5358 PRINT:PRINT
5360 S$="DURING THE PAST WEEK*":GOSUB9100
5365 IFCN>0THEN5390
5367 S$="ALL THE REMAINING CREW DIED*":GOSUB9100
5369 GOSUB9200:PRINT:GOSUB9200
5373 S$="THE SHIP DRIFTS HELPLESSLY*":GOSUB9100
5374 GOSUB9200
5376 S$="ON THE EMPTY OCEAN.....*":GOSUB9100
5378 GOSUB9200
5380 PRINT:PRINT
5382 S$="      GAME OVER*":GOSUB9100
5384 PRINT:PRINT
5386 GOSUB9200:END
5388 GOTO5386
5390 PRINTX;
5392 IF X=1 THEN S$="CREW MEMBER HAS DIED*"
5394 IFX<>1 THEN S$="CREW MEMBERS HAVE DIED*"
5396 GOSUB9100
5398 GOSUB9200
5400 PRINT:PRINT
5405 S$="YOU HAVE NOW RUN OUT OF "
5410 FOR T=1TO4
5415 IFPA(T)<>-999THEN5440
5420 PRINTS$;P$(T)
5425 PA(T)=0
5428 S$="                    AND "
5440 NEXT
5450 GOSUB9200
5455 REM CALC NEW JL
5460 X=0
5465 FORT=1TO16
5470 X=X+TS(T,2)
5475 NEXT
5480 IFX>799THEN5494
5481 PRINT:PRINT
5482 S$="THE CREW IS BELOW FULL STRENGTH*":GOSUB91
00
5483 S$="SO JOURNEY MIGHT TAKE LONGER*":GOSUB9100
5489 EW=EW+((800-X)/800)
5490 JL=JL+INT(EW)
5492 PRINT
5494 S$=K$:GOSUB9100
5496 GETI$:IFI$=""THEN5496
5499 RETURN
```

Extra Weeks Variable

```
41 EW=0: REM EXTRA WEEKS
```

Addition To Main Voyage Loop

```
880 GOSUB 5300: REM END-OF-WEEK REPORT
```

Basic Flavours

Spectrum:

Make the following changes:

```
5305 CLS
5496 LET I$=INKEY$: IF I$="" THEN GOTO 5496
```

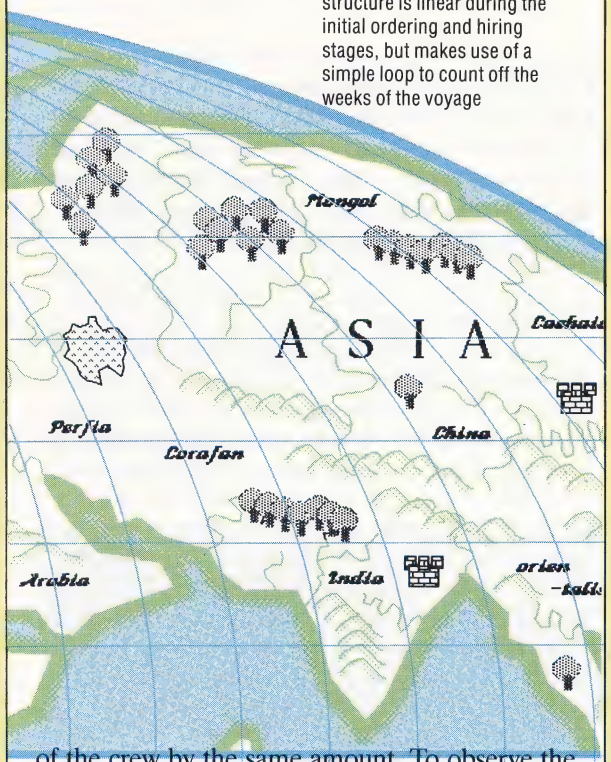
BBC Micro:

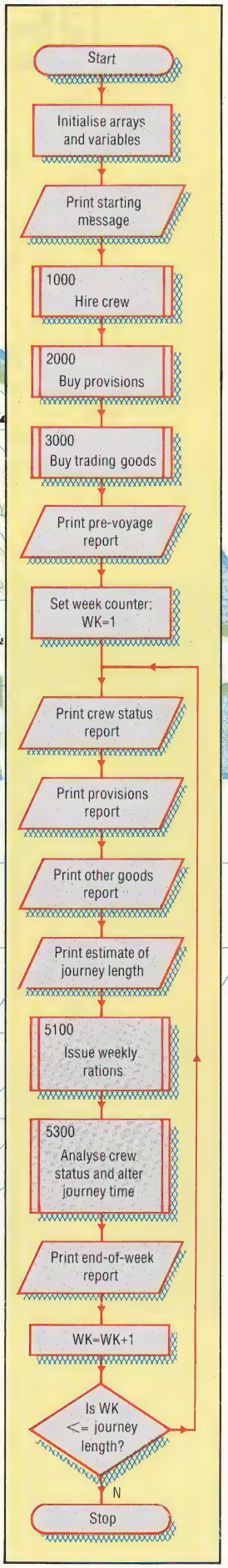
Make the following changes:

```
5305 CLS
5496 I$=GET$
```

IAN McKINNELL

The Story So Far
After Module Five of the program has been added, we can review the stages already programmed in our New World simulation game using a flowchart. The program's structure is linear during the initial ordering and hiring stages, but makes use of a simple loop to count off the weeks of the voyage





of the crew by the same amount. To observe the consequences of reducing individual strength ratings at this stage of the game, we can insert a small routine that sets the strength ratings randomly at the start of the voyage. If you use this routine, however, you must remember to remove it before typing in the next instalment of the program.

```
601 REM TEST CODE
602 FOR T = 1 TO 16
603 IF TS(T,2)=100 THEN TS(T,2)=
      INT(RND(1)*100)+1
604 NEXT
```

The routine at line 603 goes round the second element of the crew array, for 16 iterations. It changes any strength rating of 100 to a random number between 0 and 99, using the RND formula given (1 is added to each randomly generated number to avoid setting a strength rating to zero, and thus killing the crew member).

The end of week report routine is called by a GOSUB call from within the main voyage loop at lines 820 to 889 (see page 1108). Originally, this loop was controlled by a FOR . . . NEXT construction, the upper limit being set by the journey length variable, JL. However, as the new module can alter JL's value, we therefore need to reconstruct the loop by replacing the FOR and NEXT statements with:

```
820 WK=1: REM MAIN VOYAGE LOOP
889 WK=WK+1: IF WK <= JL THEN 825
```

RECORD TIME

Continuing our look at PASCAL's varied data structuring methods, we consider further operations on sets and examine another structured type — the 'record'. Records are the most useful way of packaging data of differing data types together.

In the previous instalment, we showed how sets can be extremely useful to the PASCAL programmer, and are able to be implemented efficiently at machine level because the operations defined on them have direct equivalents in most processors' native instruction sets. They are always constructed by collecting a group of elements that can be any true scalar type (not real). Apart from being able to test for membership with the operator IN, however, all operations on sets are defined on the whole structure. Although there is no selection mechanism for extracting a particular element, we can use the set inclusion operators (<=>=), and check for set equivalence ('equality') with = and <>.

When sets are manipulated by mathematicians, they can not only contain unbounded objects, but may also be considered theoretically infinite (such are the advantages of problem-solving on a piece of paper rather than on physically constrained hardware). Your implementation will place some limit on both the size and range of sets, so don't try to define ambitious types such as:

```
HugeSet = SET OF integers;
```

The lower bound of the set membership must have an ordinal value of zero or above, and the upper bound will be restricted to some absolute value — between 255 and 4,095 on microcomputer implementations. Notice that the empty set (represented literally by []) is a member of all possible sets — whatever their type. This may appear to be a hole in PASCAL's strongly typed armour, but in practice, the type of the empty set can always be deduced from the type of other sets in any expression.

Strict inclusion of true subsets cannot be obtained directly, and the operators < and > are not defined for set structures. This is for implementation reasons — most restrictions in PASCAL make good sense from the point of view of efficiency or sheer logic. Should you wish to test for strict inclusion, a dual test is needed, for example:

```
( A >= B ) AND ( A <> B )
```

The test for House in our Bingo program (see page 1116) did not require this special test, as a set of called numbers may either be equivalent ('equal') or, more likely, a superset of the numbers on the card. So, for instance, this Boolean expression could have been expressed as:

```
House := Called >= Card
```

which becomes true when all the members of Card are included in the set of Called numbers. This is exactly the sort of fundamental property of sets that makes them valuable data structures for problem-solving. For now, probably the most useful application of sets will be for testing subsets of the type char. The following scheme (in PASCAL/English) would be helpful for programming any interactive application such as a game.

```
Negative := [ 'N', 'n' ];
Affirmative := [ 'Y', 'y' ];

REPEAT
   { Play the game }
   { display the score(s) }
   write ( 'Again ? ' );
   ReadLn ( reply );

   WHILE NOT ( reply IN Affirmative + Negative ) DO
      BEGIN
         WriteLn ( 'Y(es) or N(o)' :Columns );
         write ( 'Another game ? ' );
         ReadLn ( reply )
      END

UNTIL reply IN Negative
```

When we wish to access individual elements of a structure, the choice open to us falls between arrays (familiar enough), files and records. The latter have the exceptionally powerful ability to mirror real-life records of data, with mixed 'fields' of any data type — simple or structured.

PASCAL'S USE OF RECORDS

The most common form of data record used in business contains fields for names, addresses, telephone numbers, account codes and so on. The important thing is to be able to handle this data as one lump of information and *also* to be able to access any individual field and process the data appropriately. PASCAL allows such objects to be assigned (and manipulated in other ways) as a whole, yet enables access to any component fields for comparison or processing according to their individual data type. The definition of a mixed record is quite straightforward:

```
TYPE
   room = RECORD
          number   : 1 .. 999;
          wing     : ( North, East, South, West );
          occupied : boolean
       END; { room }
VAR
   office : room;
```

Just as the CASE statement was exceptional in using the reserved word END as a delimiter, the definition

of a record is the only exception in the declaration part of a block to the rule that BEGINs and ENDs are used in pairs. For this reason it is helpful to comment a record's END with its identifier, as in our example. Any variable of type room will contain three component fields. In this case, each field is of a different scalar type, but they could just as well be the same type — simple or structured. There are no restrictions whatsoever on the types allowed within record fields, so we can have one field that might be an array of files of records containing sets!

Within the delimiting words RECORD and END, the defining syntax is exactly the same as for a VAR declaration. Here, however, we are declaring field identifiers that are an integral part of the record structure. Hence the names number, wing and occupied do not exist outside the 'scope' of the record's identifier. These field names are local identifiers, and they could duplicate the names of variables in the program. They can only be accessed via PASCAL's two selection mechanisms for records: 'dot' notation and the WITH statement.

To select a particular field using dot notation, the whole record's identifier is separated from the following field identifier by a dot ('.').
For example:

```
office . number
```

would refer to the integer subrange field only. We could initialise the contents of the record with statements such as:

```
read ( office . number );
'office . wing := East;'
'office . occupied := people <> [ ]'
```

and so on. Notice that we sneaked in an empty set with a type that depends on whatever type people is — we haven't actually declared it!

THE WITH STATEMENT

Dot notation can become somewhat cumbersome with cases in which we want to access most or all of a record's fields. An alternative structured statement is available that 'uncovers' the field identifiers. The WITH statement's semantics mean, roughly: 'I want to do something with this record, so I wish to specify only the field names'. The syntax of the WITH statement has exactly the same form as the WHILE loop (see page 1092), and the sequence of initialising assignments we have given could be better expressed as:

```
WITH office DO
  BEGIN
    number := 123;
    wing := East;
    occupied := true
  END
```

Throughout the extent of the WITH statement (from the BEGIN to its END in this case), the field identifiers do not have to be qualified with the record's identifier and dot and may be referred to directly, as shown. There is no confusion here should another program variable be named Number, for example. The local scope always takes precedence. Using dot notation, however, we can communicate values across these scope boundaries. As an example:

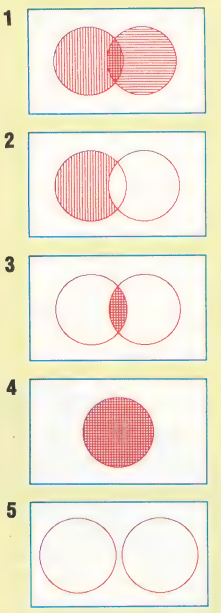
```
office . number := Number
```

would assign an external variable's value (Number) to the record field (number). Remember that

PASCAL Set Operations

This is a list of all the set operations available in PASCAL (with corresponding Venn diagrams where appropriate)

1. Union (S1 + S2): This is the superset comprising all the objects in both set S1 and set S2 combined (diagram 1)
2. Difference (S1 — S2): The subset of the members of S1 that do not also belong to S2 (diagram 2)
3. Intersection (S1 * S2): The subset of objects common to both sets (diagram 3)
4. Equivalence (S1 = S2): Where the membership of S1 and S2 is identical (diagram 4)
5. Non-equivalence (S1 <> S2): This is true when every member of S1 is not a member of S2, and vice versa (diagram 5)
6. Inclusion (S1 <= S2): This is true if every member of S1 is also a member of S2
7. Inclusion (S1>=S2): This is true if every member of S2 is also a member of S1
8. Membership (m IN S1): This is true if the single element set ([m]) is a subset of S1

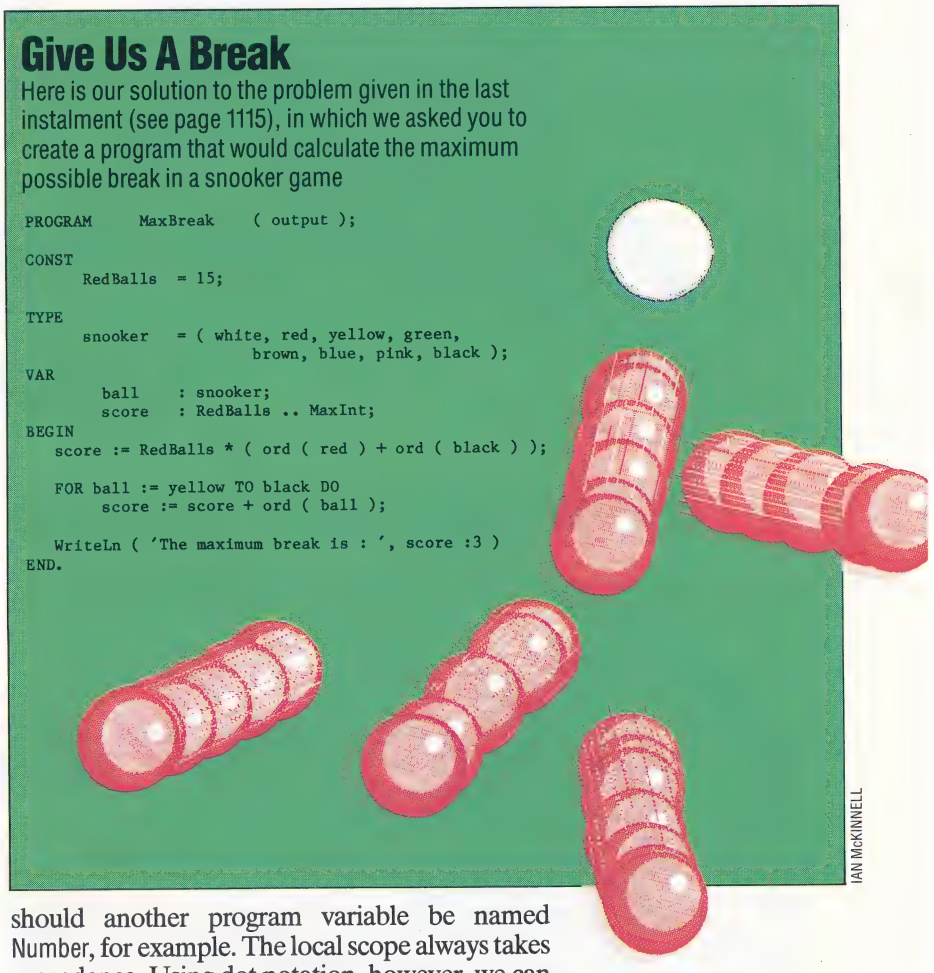


LIZ DIXON

Give Us A Break

Here is our solution to the problem given in the last instalment (see page 1115), in which we asked you to create a program that would calculate the maximum possible break in a snooker game

```
PROGRAM     MaxBreak    ( output );

CONST
     RedBalls  = 15;

TYPE
     snooker   = ( white, red, yellow, green,
                   brown, blue, pink, black );
VAR
       ball    : snooker;
       score   : RedBalls .. MaxInt;
BEGIN
  score := RedBalls * ( ord ( red ) + ord ( black ) );

  FOR ball := yellow TO black DO
      score := score + ord ( ball );

  WriteLn ( 'The maximum break is : ', score :3 )
END.
```



IAN McKINNELL

PASCAL is not case-sensitive (it doesn't distinguish between small or capital letters), and the unqualified identifier number would refer to an external variable unless it was within a WITH statement. In any case, it would be essential to use dot notation if we had an assignment between two variables of the same record type, say:

```
parcels . wing := reception . wing
```

Here, the WITH statement could only be used to uncover one of the variable's fields, otherwise there would be a confusing ambiguity, which the compiler would disallow. The best we could do would be:

```
WITH parcels DO
  wing := reception . wing
```

Consequently, both forms of notation have their uses. Which form to use will usually be obvious from the application.

In the next few instalments we shall look at arrays and files in PASCAL, and begin to investigate the full potential of these exceptionally powerful data structuring methods.

A Long Distance Run

The program Distances reads two lengths expressed in yards, feet and inches from the keyboard. A record type definition is used to allocate a separate field for each value of the differing units of measurement. The program then adds the lengths together, carrying any inches over into the field 'ft' and surplus feet into the 'yds' field by using the DIV and MOD integer operators to give the sum and remainder respectively. The results are assigned to the fields of 'Total' and then printed. Direct assignment of whole records would be straightforward (e.g: Alength := Blength) but any processing or manipulation must be applied to each component field, so that Total :=Alength + Blength is illegal. Notice the use of dot notation and the WITH statement to read the values for Alength and Blength

```
PROGRAM         Distances       ( input, output );

CONST
        MaxByte  = 255;

TYPE
        byte     = 0 .. MaxByte;

        distance = RECORD
                        ins     : 0 .. 11;
                        ft      : 0 .. 2;
                        yds     : byte

                   END; { distance }
VAR
        inches,
        feet    : byte;
        Alength,
        Blength,
        Total   : distance;

BEGIN

  WriteLn ( 'Enter lengths as yds ft and ins,' );
  WriteLn ( 'separated by  SPACE  or  RETURN.' );
  WriteLn;
  write ( 'First : ' : 15 );
  read ( Alength . yds, Alength . ft, Alength . ins );
  write ( 'Second : ' : 15 );

  WITH Blength DO
    read ( yds, ft, ins );

  inches := Alength . ins + Blength . ins;
  feet := Alength . ft + Blength . ft + inches DIV 12;
  WriteLn;

  WITH Total DO
    BEGIN
      ins := inches MOD 12;
      ft := feet MOD 3;
      yds := Alength . yds + Blength . yds
                           + feet DIV 3;
      WriteLn ( 'The total length is :' );
      WriteLn ( yds : 25, ' yds. ', ft : 1,
                ' ft. and ', ins : 1, ' ins.' )
    END
END
```

Data Protection

We set an exercise with our Bingo program (see page 1116) for you to add loop constructs to the program to prevent the repetition of a called number and detect an illegal number outside the range one to 90. The best way to protect the Bingo program from erroneous data entry is to add a WHILE construct after both ReadLn statements.The structure will be the same in each case:

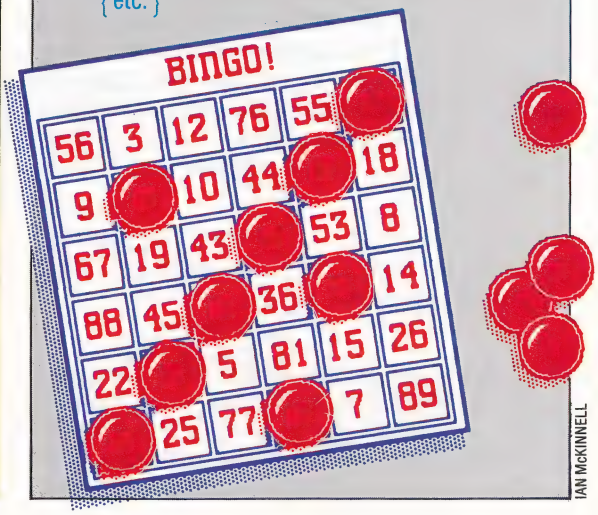
```
WHILE the number is unacceptable DO
  give a suitable error message
  write another prompt
  read the data again
```

In the case of the initial card entries, we must ensure that the entered number is not outside the range allowed (one to 90) and also prevent the repetition of a previous entry. Thus:

```
WHILE NOT (number IN Allowed) OR (number IN
Card) DO
  BEGIN
    WriteLn (number: 20, 'is not valid');
    write ('Re-enter:' :14);
    ReadLn (number)
  END;
    { etc.}
```

When calling the numbers, the same strategy applies but with the condition for entering the validation loop altered to:

```
WHILE NOT (number IN Allowed) OR (number
IN Called) DO
  { etc. }
```



IAN McKINNELL

AROUND THE BLOCK

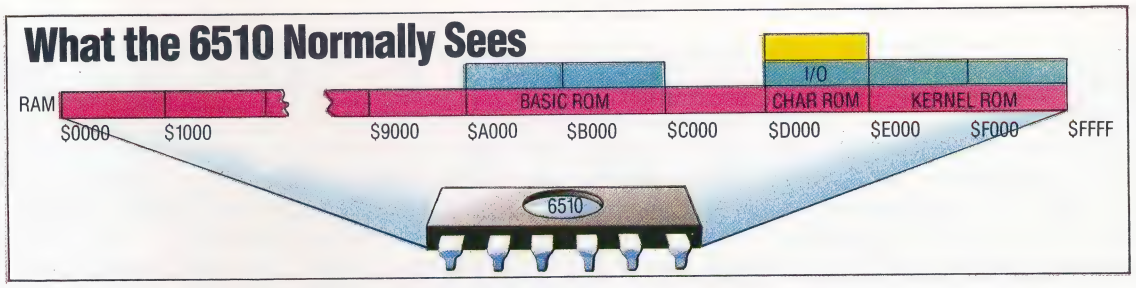
We begin an investigation of useful or unusual aspects of the Commodore 64's operating system by taking a detailed look at the first one Kbyte block of memory. Part of this block is devoted to the RAM vectors, the use of which is illustrated in our first 6510 assembler program, which creates an on-screen clock.

The first thing we need to know about any computer from a system programming point of view is: what is the overall structure of the memory map? In the case of the Commodore 64 the answer is: the 6510 microprocessor may 'see' one of *several* maps, depending on how the programmer configures the machine. This is because the 6510 is a very versatile microprocessor with the ability to 'bank switch' different blocks of memory (see page 148). This can be accomplished in hardware by bringing pins 8 and 9 on the expansion port high (+5v) or low (0v), or in software by altering the content of address 1. Either of these manoeuvres will radically alter how the 6510 views the memory. We give a skeletal picture of the normal (default) map, showing the 64 Kbytes of memory normally seen by the 6510 processor. A more detailed analysis of the memory map is given in the table.

For the time being, let's assume that the Commodore 64 is in its default mode. The first thing to note is that the computer has 64 Kbytes of RAM and 20 Kbytes of ROM. At no time can the 6510 address more than 64 Kbytes of the 84 Kbyte total, and hence the need for the memory bank switching facility. We can see from the memory map diagram that the BASIC interpreter ROM — the machine code program that runs BASIC programs — is located between $A000 and $CFFF, and the kernel ROM — the code that handles all input/output functions — is between locations $E000 and $FFFF.

Beneath these two blocks of ROM are two eight Kbyte blocks of RAM, which the 6510 does not normally see, although a POKE (or STA) to these addresses will 'drop through' to the underlying RAM. A PEEK will give the contents of the overlying ROM, but the RAM can be read by banking out the corresponding ROM. It must be remembered that if you are planning on using the BASIC PEEK or POKE commands then you had better not bank out the BASIC ROM!

Just as any BASIC program needs RAM in which to store its variables, so the interpreter and kernel routines need OS RAM. The OS RAM is found at locations $0002 to $03FF, and an important component of this, the stack, occupies locations $0100 to $01FF. The OS RAM is placed on the zero page because it is the part of the memory map that allows the fastest RAM access, and speed of operation is an important consideration for the operating system. If you plan to interface a BASIC program with some machine code, then a working knowledge of this area of RAM is essential. Let's consider two important aspects of the OS RAM: BASIC pointers and machine code vectors.



A ROM With A View

As the 6510 processor uses 16-bit addresses, it can only address a maximum of 65,536 (64K) memory locations. The Commodore 64 has a total of 84K of ROM and RAM, but gets around the problem of only being able to 'see' 64K at a time by bank switching memory areas in and out of its memory map.

The front 'block' represents the memory that the 6510 normally looks at. Areas lying behind this can be accessed by setting special banking registers. For example, if a program is written purely in machine code then the BASIC ROM may not be required and banked out so that the RAM lying behind can be used

Commodore 64 Memory Map Analysis

Locations	Use
$0000-$0001	6510 memory map control registers
$0002-$03FF	Operating System RAM
$0400-$07F7	Screen RAM
$07F8-$07FF	Sprite pointers
$0800-$9FFF	User BASIC program area (including variables)
$A000-$BFFF	BASIC interpreter ROM
$C000-$CFFF	4 Kbytes of free RAM
$D000-$D02E	VIC II (6566/9) chip control registers
$D02F-$D3FF	Repeat VIC II images
$D400-$D41C	SID (6581) chip control registers
$D41D-$D7FF	Repeat SID images
$D800-$DBE7	Colour nybbles (4 bits each)
$DBE8-$DBFF	Unused nybbles
$DC00-$DC0F	CIA #1 I/O (6526) chip control registers
$DC10-$DCFF	Repeat CIA #1 images
$DD00-$DD0F	CIA #2 I/O (6526) chip control registers
$DD10-$DDFF	Repeat CIA #2 images
$E000-$FFFF	Kernel ROM

BASIC POINTERS

Altering the contents of the BASIC pointers can be very useful at times. For example, the normal contents of $002B (decimal 43) and $002C (decimal 44) are 1 and 8 respectively. This is the address of the start of BASIC held in lo-byte/hi-byte form, and signifies that BASIC begins at 8×256+1 = 2049, or $0801. In fact, BASIC actually starts at location $0800, but the operating system requires that the first byte is always a zero, so that the BASIC program proper starts at $0801. (Incidentally, $0801 is the address at which to start saving a BASIC program when using the Commodore machine code monitor.)

Before loading any BASIC program we can alter the bottom of BASIC by manipulating these two pointers, taking care to ensure that BASIC starts at a page boundary (that is, keep the contents of location $002B at one) and starts with a zero. (Note, also, that a 'page' is a 256 byte block of memory.) Thus:

```
POKE2560,0:POKE44,10:NEW
```

executed in direct mode will raise the bottom of memory by two pages to $2560. Here, NEW is used as a quick way to reset all the pointers (occupying locations $002D to $0038). Raising the bottom of BASIC can be used to hold two BASIC programs in memory simultaneously. Simply load the first program, raise the bottom of BASIC and then load the second.

More often, however, you will want to lower the top of memory to create some space for a machine code program. Thus:

```
POKE56,159:POKE51,0:POKE52,159
```

will lower the top of BASIC memory by one page.

Once we have removed a block of RAM from the province of BASIC we can be sure that the OS will not use this area for storing BASIC variables, and therefore our machine code will be safe (except from our own mistakes!).

CBM 64 BASIC Pointer Table

Locations	Pointer
$002B-$002C	Start of user BASIC
$002D-$002E	Start of BASIC variables
$002F-$0030	Start of BASIC arrays
$0031-$0032	End of BASIC arrays + 1
$0033-$0034	Bottom of strings
$0035-$0036	Utility
$0037-$0038	End of user BASIC

THE RAM VECTORS

The second block of RAM of particular interest to the machine code programmer is the block between locations $0314 and $0333, which contains the RAM vectors. A RAM vector is like an optional switch to a branch line on a railway; usually, when a train (or, by analogy, the 6510 executing its program) comes down the line it goes over the switch without diverting its course. Sometimes, however, it is convenient to switch the train down a side line and call at one or two other stations before resuming the normal route.

Let's use the IRQ (Interrupt ReQuest) vector as an example. Every one sixtieth of a second, when the Commodore 64 is operating normally, one of the 6526 I/O chip timers is triggered and puts the 6510 IRQ line low. At the end of its current instruction the 6510 will respond to the IRQ line going low by generating an interrupt and beginning the IRQ service routine, a piece of housekeeping code that starts at $FF48. Among other things, the IRQ service routine scans the keyboard to detect any keypresses. One of the first things this routine does is:

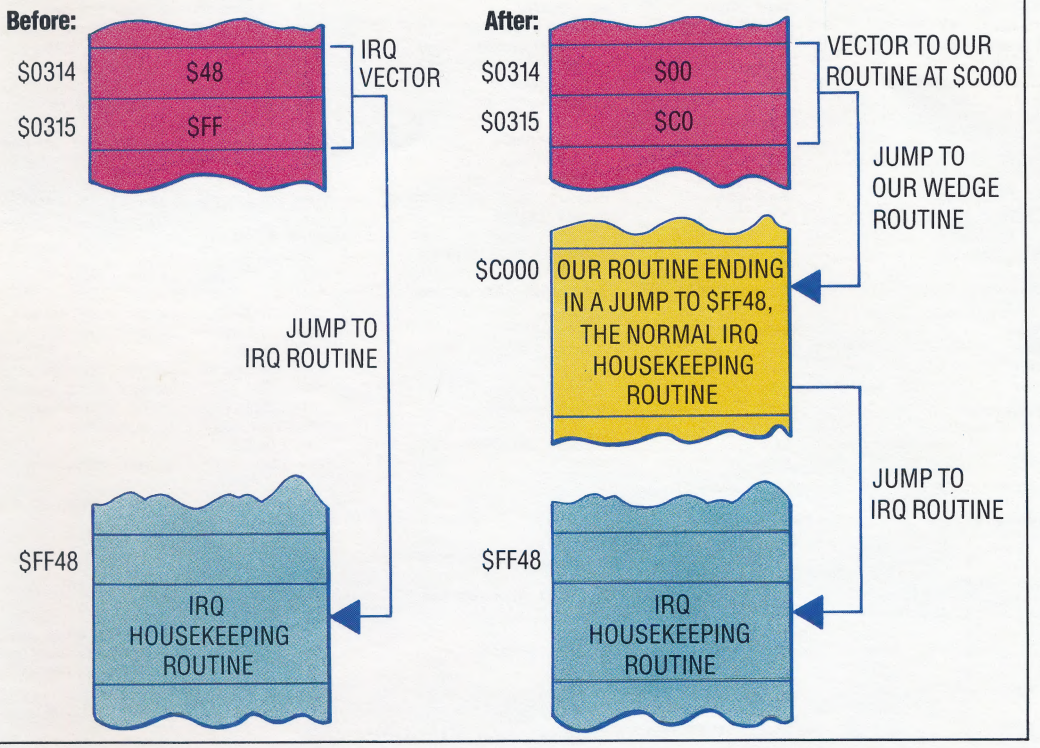
```
JMP ($0314)
```

that is, make an indirect jump to the address given in locations $0314 (lo-byte) and $0315 (hi-byte). These two bytes are in RAM, so it stands to reason that if we change their contents we can point to their own section of code. After executing our code we can point the microprocessor back in the direction it should have been going. Thus, as long as our code doesn't get too long (which precludes using any kernel routines) we can make the 6510 do a little routine for us every one sixtieth of a second. Such a piece of code is called a 'wedge'.

CBM 64 Vector Table

Locations	Vector
$0314-$0315	IRQ interrupt
$0316-$0317	BRK interrupt
$0318-$0319	NMI interrupt
$031A-$031B	Kernel OPEN routine
$031C-$031D	Kernel CLOSE routine
$031E-$031F	Kernel CHKIN routine
$0320-$0321	Kernel CHKOUT routine
$0322-$0323	Kernel CLRCHN routine
$0324-$0325	Kernel CHRIN routine
$0326-$0327	Kernel CHROUT routine
$0328-$0329	Kernel STOP routine
$032A-$032B	Kernel GETIN routine
$032C-$032D	Kernel CLALL routine
$032E-$032F	User defined
$0330-$0331	Kernel LOAD routine
$0332-$0333	Kernel SAVE routine

Driving In A Wedge



KEVIN JONES

The Wedge Of Time

The 6510 processor is interrupted every one-sixtieth of a second to do housekeeping chores such as scanning the keyboard. The start address of the housekeeping routine is held in a pair of locations — $0314 and $0315.

By changing the address held in these locations, we can 'wedge in' our own piece of code, which will be executed instead of the housekeeping routine. Usually, such a piece of wedge code ends by jumping to the normal housekeeping routine. In this way we can ensure that our code will be executed every one-sixtieth of a second, when an interrupt is generated

Commodore Clock

The following assembly listing and BASIC loader program demonstrates the principles of inserting a piece of 'wedge' code into the normal IRQ interrupt handling routine. This wedge runs a 'time of day' clock in the corner of the screen. As the clock is updated by the normal interrupt, which occurs every one sixtieth of a second, the machine will operate normally with the clock running, allowing a BASIC program to be typed in and run simultaneously. The routine first sets up the initial times passed to it, and then changes the contents of $0314 and $0315 to point at the wedge code, which updates and displays the clock each time an IRQ interrupt is generated. The final task is to make a jump (JMP) to the normal IRQ housekeeping routine, the address of which is preserved in locations VECTOR and VECTOR+1.

As an alternative to typing in and assembling this source code, you can enter the program as a series of DATA statements that form the machine code program. Simply type in the BASIC loader program and RUN to see the clock appear in the top right corner of the screen. A checksum is included to ensure that all the DATA has been entered correctly. If the program stops with the message 'CHECKSUM ERROR' check your DATA statements for a mistake.

The clock can be set by POKEing in the starting times in hours, minutes and seconds into locations 50129, 50130 and 50131. At the end of the BASIC loader program the clock is set up by lines 1480 onwards to start at 4:30:00pm. Simply change the values POKEd into these locations to alter the start time accordingly. SYS 50138 causes the machine code program to be executed and the clock to be started. The clock can be turned off by pressing the RUN/STOP and RESTORE keys simultaneously, or by executing a special turn-off routine included in the machine code program by entering SYS 50237

Basic Loader

```
1000 REM ** CLOCK BASIC LOADER **
1010 DATA173,166,2,240,10,169,128,13,14
1020 DATA221,141,14,221,48,8,169,127,45
1030 DATA14,221,141,14,221,169,127,45
1040 DATA15,221,141,15,221,173,208,195
1050 DATA41,128,141,208,195,173,209,195
1060 DATA32,28,197,13,208,195,141,11
1070 DATA221,173,210,195,32,28,197,141
1080 DATA10,221,173,211,195,32,28,197
1090 DATA141,9,221,169,0,141,8,221,120
1100 DATA173,20,3,141,214,195,173,21,3
1110 DATA141,215,195,169,76,141,20,3
1120 DATA169,196,141,21,3,88,96,120,173
1130 DATA214,195,141,20,3,173,215,195
1140 DATA141,21,3,88,96,173,216,195,201
1150 DATA6,240,3,76,8,197,169,255,141
1160 DATA216,195,173,213,195,240,243
1170 DATA173,11,221,170,41,128,208,5
1180 DATA169,1,76,111,196,169,16,141,38
1190 DATA4,173,212,195,141,38,216,163
1200 DATA13,141,39,4,173,212,195,141,39
1210 DATA216,138,41,16,32,14,197,141,28
1220 DATA4,173,212,195,141,28,216,138
1230 DATA32,22,197,141,29,4,173,212,195
1240 DATA141,29,216,169,58,141,30,4,173
1250 DATA212,195,141,30,216,173,10,221
1260 DATA170,32,14,197,141,31,4,173,212
1270 DATA195,141,31,216,138,32,22,197
1280 DATA141,32,4,173,212,195,141,32
1290 DATA216,169,47,141,33,4,173,212
1300 DATA195,141,33,216,173,9,221,170
1310 DATA32,14,197,141,34,4,173,212,195
```

```
1320 DATA141,34,216,138,32,22,197,141
1330 DATA35,4,173,212,195,141,35,216
1340 DATA169,46,141,36,4,173,212,195
1350 DATA141,36,216,173,8,221,105,48
1360 DATA141,37,4,173,212,195,141,37
1370 DATA216,238,216,195,108,214,195,74
1380 DATA74,74,74,24,105,48,96,41,15,24
1390 DATA105,48,96,160,255,56,200,233
1400 DATA10,176,251,105,10,141,217,195
1410 DATA152,10,10,10,10,13,217,195,96
1420 DATA42131:REM*CHECKSUM*
1430 CC=0
1440 FORI=50138TO50481
1450 READX:CC=CC+X:POKEI,X
1460 NEXT
1470 READX:IFCC<>XTHENPRINT"CHECKSUM ERROR"
1480 REM** TEST CLOCK **
1490 POKE50128,128:REM AM/PM
1500 POKE50132,8:REM COLOUR
1510 POKE50133,1:REM DISPLAY
1520 POKE50129,4:REM HOURS
1530 POKE50130,30:REM MINUTES
1540 POKE50131,0:REM SECONDS
1550 SYS50138:REM CALL ROUTINE

+O TURN OFF THE WEDGE USE SYS50237
IN DIRECT OR PROGRAM MODE.
```

Assembly Listing

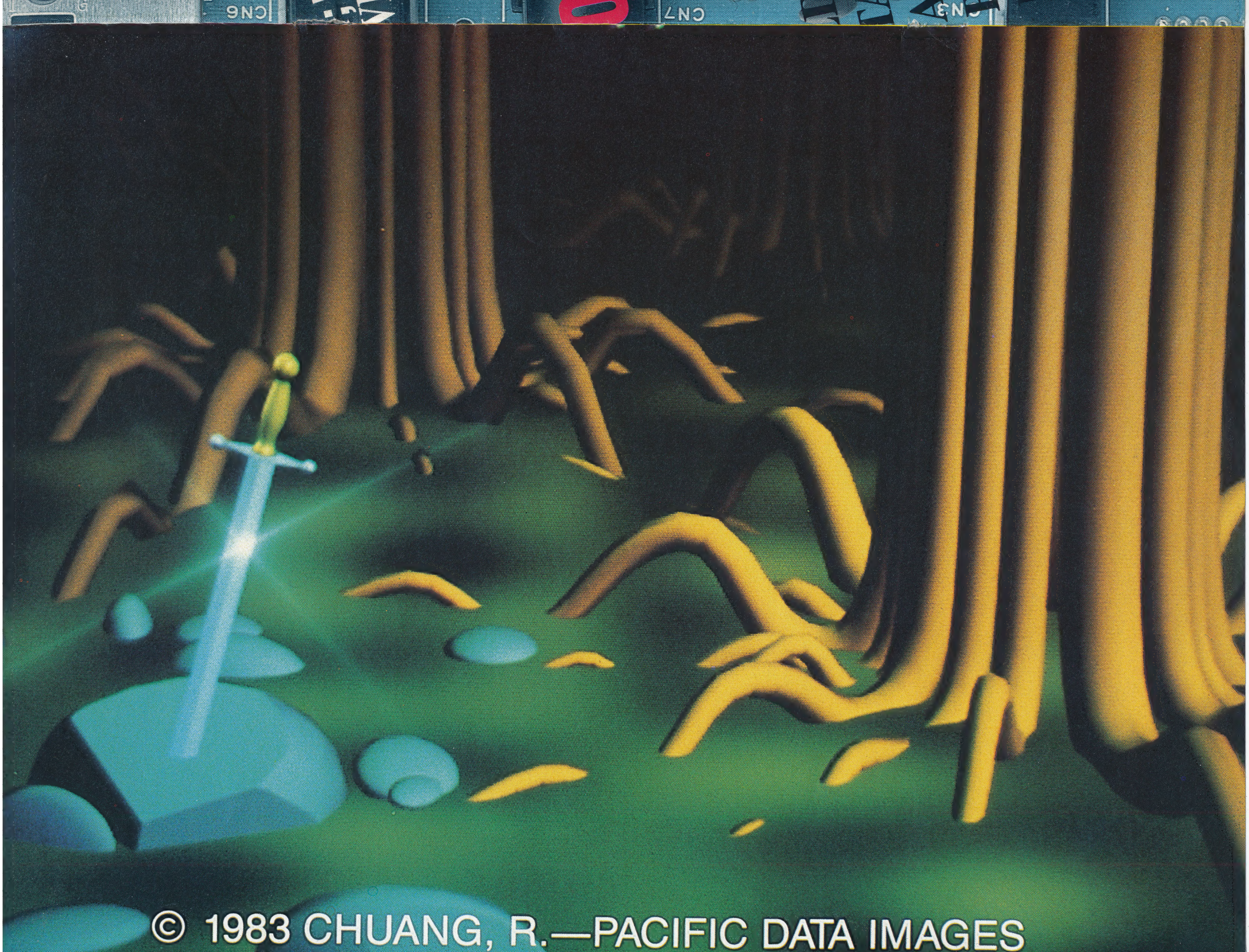
```
;***********************************
;*                                 *
;*        CLOCK IRQ WEDGE          *
;*                                 *
;* 50128 = AM=0 / PM=128           *
;* 50129 = HOURS                   *
;* 50130 = MINUTES                 *
;* 50131 = SECONDS                 *
;* 50132 = CLOCK COLOUR            *
;* 50133 = DISPLAY ON=1/OFF=0      *
;*                                 *
;* WEDGE INSERT  SYS 50138         *
;* WEDGE REMOVE  SYS 50237         *
;*                                 *
;***********************************

IRQVEC = $0314   ; IRQ RAM VECTOR
CLOCK = $DD08    ; TOD REGISTER
D2CRA = $DD0E    ; VIA#2 CRA
D2CRB = $DD0F    ; VIA#2 CRB
PALNTS = $02A6   ; PAL/NTSC FLAG
RATE = $06       ; DISPLAY EVERY 6 IRQS
DIGIT = $30      ; SCREEN CODE FOR '0'
POINT = $2E      ; SCREEN CODE FOR '.'
SLASH = $2F      ; SCREEN CODE FOR '/'
COLON = $3A      ; SCREEN CODE FOR ':'
HZ50 = $80       ; 50 HZ MASK FOR TODIN
HZ60 = $7F       ; 60 HZ MASK FOR TODIN
AY = $01         ; SCREEN CODE FOR 'A'
PEE = $10        ; SCREEN CODE FOR 'P'
EM = $0D         ; SCREEN CODE FOR 'M'
WRITE = 127      ; MASK TO SET CLOCK IN CRB
SCNLOC = $041C   ; CLOCK ADDR ON SCREEN
COLLOC = $D81C   ; ADDR ON VIDEO MATRIX
TRNCLO = $0F     ; MASK FOR LOW NYBBLE

* = $C300        ; START ADDRESS CODE
AMPM *=*+1       ; AM/PM FLAG
HOURS *=*+1      ; HOURS VAL (FOR INITIALISE)
MINS *=*+1       ; MINUTES VALUE
SECS *=*+1       ; SECONDS VALUE
COLOR *=*+1      ; CLOCK COLOUR
DISPLY *=*+1     ; DISPLAY/HIDE FLAG
VECTOR *=*+2     ; STORAGE FOR OLD IRQ VEC
COUNT *=*+1      ; IRQ COUNTER
TEMP1 *=*+1

;
;INSERT WEDGE
;
   LDA PALNTS      ; PAL OR NTSC
   BEQ NTSC        ; BRANCH FOR NTSC
   LDA #HZ50       ; MUST BE PAL
   ORA D2CRA
   STA D2CRA       ; SET TOSIN FOR 50 HZ
   BMI PALDUN
NTSC
   LDA #HZ60       ; NTSC
   AND D2CRA
   STA D2CRA       ; SET TODIN FOR 60 HZ
PALDUN
   LDA #WRITE
   AND D2CRB       ; SET CLOCK NOT ALARM
   STA D2CRB
   LDA AMPM
   AND #128        ; MAKE AMPM VALUE VALID
   STA AMPM
   LDA HOURS       ; GET HOURS
   JSR BINBCD      ; CONVERT TO BCD
   ORA AMPM        ; OR WITH AM/PM FLAG
   STA CLOCK+3     ; STORE IN CLOCK
   LDA MINS        ; GET MINUTES
   JSR BINBCD      ; CONVERT TO BCD
   STA CLOCK+2     ; STORE IN CLOCK
   LDA SECS        ; GET SECONDS
   JSR BINBCD      ; CONVERT TO BCD
   STA CLOCK+1     ; STORE IN CLOCK
   LDA #00         ; ALWAYS SET 10THS TO 0
   STA CLOCK       ; START CLOCK
;
   SEI             ; DISABLE INTERRUPTS
   LDA IRQVEC
   STA VECTOR      ; SAVE OLD IRQ VECTOR
   LDA IRQVEC+1
   STA VECTOR+1
;
   LDA #<WEDGE
   STA IRQVEC
   LDA #>WEDGE     ; INSERT WEDGE
   STA IRQVEC+1
   CLI             ; ENABLE INTERRUPTS
   RTS

;
;REMOVE WEDGE
;
   SEI             ; DISABLE INTERRUPTS
   LDA VECTOR
   STA IRQVEC      ; RESTORE RAM VECTOR
   LDA VECTOR+1
   STA IRQVEC+1
   CLI             ; ENABLE INTERRUPTS
   RTS

;
;WEDGE STARTS HERE
;
WEDGE
   LDA COUNT       ;
   CMP #RATE       ; DO CLOCK THIS IRQ?
   BEQ CONT        ;
OUT
   JMP EXIT        ; NO
CONT
   LDA #$FF        ; RESET IRQ COUNTER
   STA COUNT
   LDA DISPLY      ; DISPLAY?
   BEQ OUT         ; NO...BRANCH

   LDA CLOCK+3     ; GET HOURS/AM/PM
   TAX             ; PUT A COPY IN X REG
   AND #$80        ; GET AM/PM
   BNE PM          ; BRANCH IF PM
   LDA #AY         ; DISPLAY 'A'
   JMP MERIDP
PM
   LDA #PEE        ; DISPLAY 'P'
MERIDP
   STA SCNLOC+10
   LDA COLOR       ; GET COLOUR
   STA COLLOC+10   ; SET COLOUR
   LDA #EM         ; DISPLAY 'M'
   STA SCNLOC+11
   LDA COLOR       ; SET COLOUR
   STA COLLOC+11
;
;DO HOURS
;
   TXA             ; GET HOURS
   AND #$10        ; JUST WANT HIGH DIGIT
   JSR HIDIGT      ; GET SCREEN CODE
   STA SCNLOC      ; DISPLAY IT
   LDA COLOR
   STA COLLOC      ; SET COLOUR
   TXA             ; GET BYTE AGAIN
   JSR LODIGT      ; GET LOW DIGIT
   STA SCNLOC+1    ; DISPLAY IT
   LDA COLOR
   STA COLLOC+1    ; SET COLOUR
;
   LDA #COLON      ; HRS/MINS SEPARATOR
   STA SCNLOC+2
   LDA COLOR
   STA COLLOC+2
;
;NOW DO MINUTES
;
   LDA CLOCK+2     ; GET MINUTES
   TAX
   JSR HIDIGT      ; DO HIGH DIGIT
   STA SCNLOC+3    ; DISPLAY IT
   LDA COLOR
   STA COLLOC+3    ; AND COLOUR
   TXA             ; GET BYTE AGAIN
   JSR LODIGT      ; DO LOW DIGIT
   STA SCNLOC+4    ; DISPLAY IT
   LDA COLOR
   STA COLLOC+4    ; SET COLOUR
;
   LDA #SLASH      ; MIN/SEC SEPARATOR
   STA SCNLOC+5
   LDA COLOR
   STA COLLOC+5
;
;NOW DO SECONDS
;
   LDA CLOCK+1     ; GET SECONDS
   TAX
   JSR HIDIGT      ; DO HIGH DIGIT
   STA SCNLOC+6    ; DISPLAY IT
   LDA COLOR
   STA COLLOC+6    ; AND COLOUR
   TXA             ; GET BYTE AGAIN
   JSR LODIGT      ; DO LOW DIGIT
   STA SCNLOC+7    ; DISPLAY IT
   LDA COLOR
   STA COLLOC+7    ; AND COLOUR
;
   LDA #POINT      ; SECS/TENTHS SEPARATOR
   STA SCNLOC+8
   LDA COLOR
   STA COLLOC+8
;
;NOW DO TENTHS
;
   LDA CLOCK       ; GET TENTHS VALUE
   ADC #DIGIT      ; ADD $30 FOR SCREEN CODE
   STA SCNLOC+9    ; DISPLAY IT
   LDA COLOR
   STA COLLOC+9    ; AND COLOUR
;
;
EXIT
   INC COUNT       ; INCREMENT IRQ COUNTER
   JMP (VECTOR)    ; GO TO REST OF IRQ

;
;SUBROUTINES
;
HIDIGT
   LSR A
   LSR A           ; MOVE HIGH NYBBLE INTO LOW
   LSR A
   LSR A
   CLC
   ADC #DIGIT      ; ADD $30 FOR SCREEN CODE
   RTS

LODIGT
   AND #TRNCLO     ; MASK OFF HIGH NYBBLE
   CLC
   ADC #DIGIT      ; ADD $30 FOR SCREEN CODE
   RTS

;
;CONVERT BINARY TO BCD
;
BINBCD
   LDY #$FF
   SEC
D10
   INY
   SBC #10         ; SUBTRACT 10 UNTIL -VE
   BCS D10
   ADC #10         ; ADD 10 BACK ON
   STA TEMP1       ; STORE REMAINDER
   TYA             ; GET NUMBER PF 10S SUBTRACTED
   ASL A
   ASL A
   ASL A           ; SHIFT INTO HIGH NYBBLE
   ASL A
   ORA TEMP1       ; PUT REMAINDER IN LOW NYBBLE
   RTS
```

Reprinted by kind permission of the authors and Ellis Horwood Ltd. From: Mastering the Commodore 64 by Jones & Carpenter

DATABASE

Here, courtesy of Acorn Computers, we publish another instalment in a series listing the call addresses of the BBC Micro's basic ROM routines. Note that these addresses are for BASIC II only

Evaluate expression	9B29
Evaluate integer expression	8821
Evaluate operand	ADEC
Exchange FAC#1 and (&4B)	A4D6
Exponentiation routine	9E35
Expression evaluator (integer)	8821
FALSE function routine	AECA
FN function routine	B195
FOR statement routine	B7C4
Find ERL	B3C5
Fix FAC#1	A3FE
Floating point addition	9C8B
Floating point comparison	9A50
Floating point multiplication	9D20
Floating point subtraction	9CE1
GCOL statement routine	937A
GET function routine	AFB9
GET$ function routine	AFBF
GOSUB statement routine	B888
GOTO statement routine	B8CC
Get a variable name	95DD
Get an array/PROC/FN name	9559
Get character from PTR#1	8A97
Get character from PTR#2	8A8C
Get file handle	BFA9
Get the value of a variable	B32C
HIMEM function routine	AF03
HIMEM statement routine	925D
IF statement routine	98C2
INKEY function routine	ACAD
INKEY$ function routine	B026
INPUT statement routine	BA44
INPUT# statement routine	B9CF
INSTR function routine	ACE2
INT function routine	AC78
Increment MAN#1	A4B6
Increment the 1AC	9222
Initialisation of language	8023
Insert a line into program	BC8D
Integer addition	9C53
Integer comparison	9AA2
Integer division	99BE
Integer multiplication	9D41
Integer subtraction	9CBA
Keyword action addresses (LSB)	836D
Keyword action addresses (MSB)	83E0
Keyword table	8071
LEFT$ function routine	AFCC
LEN function routine	AED1
LET statement routine	8BE4
LEVEL 1 of expression evaluator	9B29
LEVEL 2 of expression evaluator	9B72
LEVEL 3 of expression evaluator	9B9C
LEVEL 4 of expression evaluator	9C42
LEVEL 5 of expression evaluator	9DD1
LEVEL 6 of expression evaluator	9E20
LIST command routine	B59C
LISTO command routine	B58A
LN function routine	A7FE
LOAD command routine	BF24
LOCAL statement routine	9323
LOG function routine	ABA8
LOMEM function routine	AEFC
LOMEM statement routine	926F
Label definition in assembler	85A5
Line number decoding	97EB
Line number encryption	88F3
Line numbers; convert to binary	889C
Line numbers; tokenise	88DB
Load a new program	BE62
MID$ function routine	B039
MOD operator routine	9E01
MODE statement routine	939A
MOVE statement routine	93E4
Mnemonic table (LSB)	8451
Mnemonic table (MSB)	848B
Move to next statement	9880
NEW command routine	8ADA
NEXT statement routine	B695
NOT function routine	ACD1
Normalise FAC#1	A303
OLD command routine	8AB6
ON ERROR OFF statement routine	B8E4
ON ERROR statement routine	B8F2
ON statement routine	B915
OPENIN function routine	BF78
OPENOUT function routine	BF7C
OPENUP function routine	BF80
OR operator routine	9B3A
OSCLI statement routine	BEC2
Op-code base table	84C5
PAGE function routine	AEC0


© 1983 CHUANG, R.—PACIFIC DATA IMAGES